UNIVERSITÀ DI PISA
Scuola di Dottorato in Ingegneria "Leonardo da Vinci"

Corso di Dottorato di Ricerca in
INGEGNERIA DELL'INFORMAZIONE

Tesi di Dottorato di Ricerca

Enforcing Application Security on Android Mobile Devices

Andrea Saracino

Anno 2015

UNIVERSITÀ DI PISA

Scuola di Dottorato in Ingegneria "Leonardo da Vinci"

Corso di Dottorato di Ricerca in
Ingegneria dell'Informazione
Tesi di Dottorato di Ricerca

Enforcing Application Security on Android Mobile Devices

Autore:

Andrea Saracino

Relatori:

Prof. Gianluca Dini *Firma*_____

Dott. Fabio Martinelli *Firma*_____

Prof. Enzo Mingozzi *Firma*_____

Anno 2015

Sommario

La sicurezza dei dispositivi mobili di nuova generazione è al momento un problema di massima importanza. Negli ultimi anni, smartphone e tablet sono diventati estremamente popolari, soprattutto nei paesi sviluppati, dove i dispositivi di nuova generazione ammontano al 95% del totale di tutti i dispositivi mobili. Data la loro popolarità, questi dispositivi hanno rapidamente attirato l'attenzione di sviluppatori malevoli che hanno iniziato a implementare e distribuire applicazioni capaci di danneggiare la privacy dell'utente, il suo denaro e persino l'integrità del dispositivo e dei dati contenuti. Questi sviluppatori hanno astutamente sfruttato la semplicità dei meccanismi di distribuzione delle applicazioni, la sensibilità delle informazioni memorizzate e delle operazioni accessibili tramite un dispositivo mobile, insieme alla limitata attenzione dell'utente medio ai problemi di sicurezza. Questa tesi presenta studio, progetto e implementazione di un sistema di sicurezza a più componenti per il popolare sistema operativo per dispositivi mobili Android. L'obiettivo di questa tesi è di fornire uno strumento di sicurezza leggero, usabile, modulare ed estensibile, che sia in grado di contrastare le minacce alla sicurezza di Android, presenti e future. Il sistema sfrutta metodologie di white-listing per identificare a tempo di esecuzione comportamenti malevoli delle applicazioni, senza essere prono al problema degli zero-day-attacks, ossia le nuove minacce non ancora identificate. L'approccio white-list è accoppiato a un approccio black-list, che riduca la possibilità di falsi allarmi e che sia in grado di contrastare comportamenti dannosi noti, prima che possano avere effetto. Il sistema proposto combina elementi di analisi statica e dinamica sfruttando elementi di teoria dei contratti, di cui si propone un'implementazione probabilistica e l'intercettazione di eventi a livello nucleo e API. Inoltre, il sistema proposto è configurabile, in modo da poter essere totalmente trasparente all'utente, o che abbia un'interazione più marcata,

quando l'utente è maggiormente interessato ad un certo livello di consapevolezza della sicurezza sul suo dispositivo. Il sistema proposto è stato testato su più di 12000 applicazioni che includono anche due grandi archivi di applicazioni malevole. I risultati di detection (95%) e sui falsi positivi (1 al giorno) provano l'efficacia del sistema. Inoltre uno studio sull'usabilità che include il feedback di 200 utenti, mostra il basso overhead (4% di batteria 1.4% di performance) e la generale accettazione degli utenti.

Abstract

Security in new generation mobile devices is currently a problem of capital importance. Smartphones and tablets have become extremely popular in the last years, especially in developed country where smartphones and tablets account for 95% of active mobile devices. Due to their popularity, these devices have fast drawn the attention of malicious developers. Attackers have started to implement and distribute applications able to harm user's privacy, user's money and even device and data integrity. Malicious developers have cleverly exploited the simplicity of app distribution, the sensitivity of information and operation accessible through mobile devices, together with the user limited attention to security issues. This thesis presents the study, design and implementation of a multi-component security framework for the popular Android operative system. The aim of this thesis is to provide a lightweight and user friendly security tool, extensible and modular, able to tackle current and future security threats on Android devices. The framework exploits white list-based methodologies to detect at runtime malicious behaviors of application, without being prone to the problem of zero-day-attacks (i.e. new threats not yet discovered by the community). The white-list approach is combined with a black-list security enforcement, to reduce the likelihood of false alarms and to tackle known misbehaviors before they effectively take place. Moreover the framework also combines static and dynamic analysis. It exploits probabilistic contract theory and app metadata to detect dangerous applications before they are installed (static analysis). Furthermore, detects and stop malicious kernel level events and API calls issued by applications at runtime (dynamic analysis), to avoid harm to user and her device. The framework is configurable and can be both totally transparent to the user, or have a stronger interaction when the user is more interested in a security awareness of her device. The presented security

framework has been extensively tested against a testbed of more than 12000 applications including two large Android malware databases. Detection rate (95%) and false positive rate (1 per day) prove the effectiveness of the presented framework. Furthermore, a study of usability which includes energy evaluation and more than 200 user feedback is presented. These results show both the limited overhead (4% battery, 1.4% performance) imposed by the framework and the good user acceptance.

To those who dream and follow their dreams always, no matter what...
i.e. my Love and my Family.

Contents

List of Figures

List of Tables

1

Introduction

Smartphones and tablets changed in the last ten years the way users access the Internet, web services and multimedia contents. The technological evolution of the data connection, which gradually moved from the 2 Mbytes of UMTS, to more than 1 GBytes with LTE advanced, has been paralleled by a strong evolution of the components and capabilities of the physical devices.

In 2008 it was correct to consider a mobile device, such as a smartphone or a tablet, as constrained. However, the continuous evolution brought in the last two years smartphones off-the-shelf whose features surpass the ones of an average desktop computer.

Hence, the features that best describe smartphones and tablets produced from 2012 to now, shape a scenario different from the one of some years ago:

- *Device highly connected*: current mobile devices have data connection (UMTS or LTE), WiFi interface, Bluetooth, Near Field Communication (NFC) and infrared. The presence of these interfaces ensure a seamless Internet connection, together with the possibility to connect and communicate directly with other devices.
- *High performance devices*: The computational power of current mobile devices is comparable with average computers. As an example, the smartphone *Motorola Nexus 6* comes with a 2.7 GHz eight-cores CPU and 8 GB of RAM.
- *Highly Customizable*: Operative systems for mobile devices allows the installation of third party applications. Thanks to third party apps, mobile phones can currently be used to access the Internet, play video-games with 3D graphics and realistic physics [3], receiving driving directions, access social network and publish multimedia live contents, or even developing new applications.

Smartphones and tablets have usage possibilities which supersede the ones of desktop and laptop computers. In particular we list here a set of common smartphone and tablets usage, with reference, when possible, to some common applications providing the described service.

- Access email server or mailbox (IMAP, POP3) or email cloud services (e.g. GMail).
- Create and edit `Office` documents, including databases, spreadsheets and presentations.
- Acquire, watch, edit or stream videos (YouTube).
- Acquire, edit and share pictures (Instagram, Facebook, Tumblr).
- Access maps and receive driving directions or location based information, also exploiting the several geo-localization interfaces (Maps, Navigator, FourSquare, TripAdvisor).
- Can be used to buy products online or perform home banking operations (Amazon, Ebay).
- Communicate with other people through instant message programs or VoIP-like systems. (Whatsapp, Yahoo, Viber, Skype)

The aforementioned functionalities, while increasing the usage possibilities for mobile devices, also bring intrinsics and serious security issues concerning user's privacy and money, together with device and data integrity.

Addressing these security issues is of capital importance. This thesis describes the design and implementation of a security framework for mobile devices to tackle attacks brought through malicious applications on Android devices.

1.1 Motivation

In this section, we will explain in detail what motivates the performed research, describing the timeliness and impact, and justifying the choice of implementing the solution on the Android system, which is currently the most popular among mobile operative systems.

At the end of 2014 the number of active mobile subscriptions all over the world is of almost 7 billions [6], where more than 2 billions of these subscriptions are for 3G (third generation connectivity: UMTS, HSPA, HSPA+)or 4G (fourth generation connectivity: LTE, LTE+) mobile contracts [6]. More precisely, in developed countries there are in average 1.2 mobile devices per person and more than 90% of mobile users have a smartphone or tablet [6, 8].

Given the strong penetration and popularity, unfortunately smartphones and tablets have become in a few years a main target for attackers. Though first attacks to 3G mobile devices have been performed starting from 2004 [88], the situation has become dramatic only in the last years. In 2012, in fact, the mobile security scenario has experienced an increase of 600%, with respect to the previous year, of attacks targeting smartphones and tablets [38]. In the specific, Figure 1.1 de-

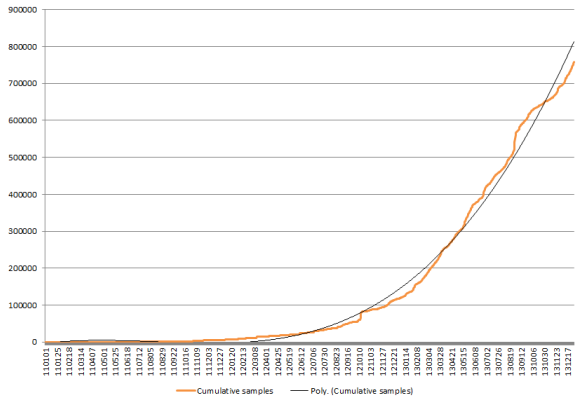


Figure 1.1: Increase of mobile malware samples between 2011 and 2012 measured by Avast [38] (x-axis date format: `yymmdd`).

picts this sudden increase of mobile threats, showing the change in the number of mobile malware samples, which has been experienced during 2012. As will be discussed in the following, mobile malware is practically the only relevant attack vector threatening mobile devices [88]. As shown in the graph of Figure 1.1, till 2011 the amount of mobile malware samples available was negligible, mainly constituted of proof-of-concepts designed by the research community to show vulnerabilities of mobile devices [88]. Instead, starting from 2012, the amount of mobile malware samples started a sharp raise. Moreover, as shown in the graph of Figure 1.2, this increasing trend in mobile malware samples has continued also for the four quarters of 2013. Now, at the end of 2014, there are more than 1 million of malicious applications which could be found in the wild, where more than 98% of these malware samples target the Android system [37].

Android is an operative system for mobile devices. The Android project is mainly controlled by Google and at the current time is the most popular operative system for mobile devices, with a share greater than 80% (see Figure 1.3). As depicted in the graph of Figure 1.3 currently the main players on the mobile market

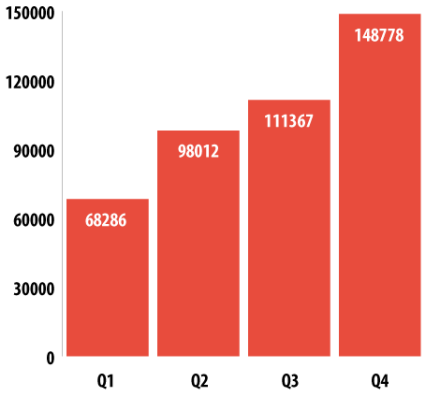


Figure 1.2: Mobile malware samples identified by Kaspersky [37] in 2013.

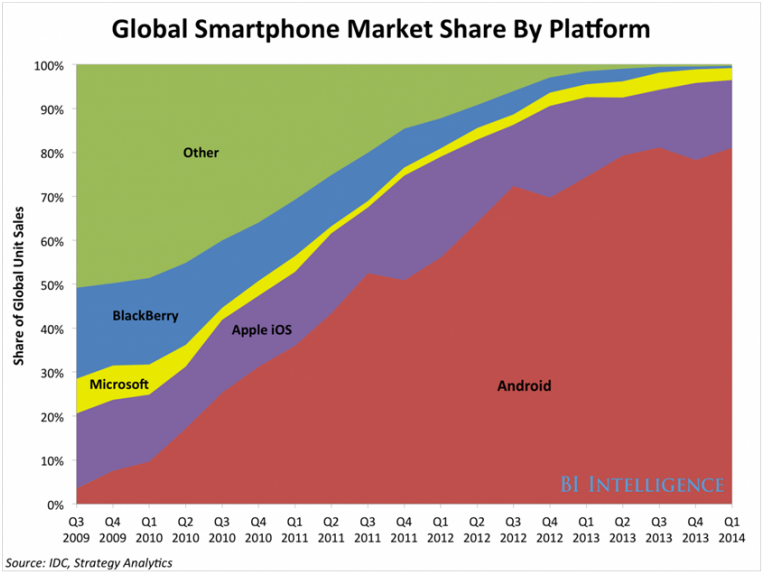


Figure 1.3: Eveolution of mobile OS market share

are in order: Android, iOS (Apple) and Windows (Microsoft). All these systems offer the possibility of installing mobile applications (apps), which are distributed through online marketplaces. Marketplaces collects applications published by developers.

Then mobile users can download published apps directly from the market, without a direct interaction with the developers.

Mobile markets strongly improve the access to mobile applications and at the same time increase the overall security, providing a more protected environment compared to the wild web, where there is no control on the identity of the app developers. However, unfortunately, also mobile market are far from being totally secure [90]. Malicious apps, in fact, have been found in these years also on these controlled channels, for example more than 15000 malicious apps have been found up to 2013 in the official market for Android applications: Google Play [90].

As a matter of fact, starting from 2012, Android has become almost the exclusive target for mobile attacks [37] with more than 98% of attacks targeting this system. The main reasons are the large popularity, i.e. greater number of possible victims, and the open source nature of the Android project. More specifically, Android code is totally available, thus it is easier to find vulnerabilities which could be maliciously exploited. Other reasons have to be found in the app distribution philosophy of Android, which privileges flexibility to strong security controls, making easier for attackers to distribute malware through non official marketplaces and even through the official one. Thus, more than 1/3 of the current Android user, have already encountered at least a malicious application until now [37].

Though simple, malicious applications are extremely dangerous and capable of seriously harming both the device and its user. In fact, cleverly exploiting the various interfaces and functionalities of a smartphone or tablet, attackers can [135]:

- Steal user private information: Mobile devices store the user's contacts, private messages (SMS or instant messaging), emails, social network account data and even bank account or credit card numbers for home banking operations.
- Steal user money: Phone calls and SMS text messages are paid services. Attackers may impose a direct loss of money to the user sending background text messages in large amounts (e.g. messages for SPAM purposes), or to register to paid premium services (e.g. subscription to weekly multimedia downloads). Most of these attacks are performed in background and the user is unlikely to notice them before it is too late.
- Hijacking bank transactions: Some malicious apps have been found to be able to intercept some home banking operations, forcing the user to authorize money transfers to account controlled by the attackers.
- Violate the device integrity: Malicious apps can access the device storage and then add, modify or delete stored data.
- Take control of the attacked device: Some malicious applications, once installed, open a back-door for attackers. This backdoor can be used to use

the attacked device as part of a botnet, or to install additional malicious code for other attacks. Often, once a device is under the control of an attacker, data wiping and factory reset are the only solutions to restore the normal functionality. It is worth mentioning that such operations are dangerous and may result in device permanent damages[1].

Given the consistent threat represented by malicious applications, several security solutions have been proposed, both by mobile producers and research community. Notwithstanding, the problem is still far to be solved, for different reasons. Malicious applications are hard to be discovered, since they come disguised as genuine ones. These apps works correctly, providing a useful service to the non suspicious user, however, they run malicious code in background. These applications are referred as *repackaged* or *trojanized*. Moreover, the malicious behaviors performed by applications are hard to detect. In fact, malicious apps maliciously exploits operations that are legitimate. For example, sending a text message, it is not by itself a malicious operation. However, sending stealthy text messages to premium numbers to perform hidden registration to paid services is a malicious behavior. Distinguishing between these two behaviors automatically however it is not trivial [112]. To detect these misbehaviors it is necessary an approach which considers together different parameters such as required authorizations and methods invocation, which are extracted analyzing the application behavior both statically and at runtime. Such an approach allows the definition of complex security policies that, when enforced, should prevent and stop app misbehaviors.

Finally, another issue is introduced by the weakest ring of the security chain, which is the user [70]. Current Android security mechanisms are mainly based on a preventive approach, that show to users the potential threats represented by new apps. However, several users, often skip the security warning of Android, either because they do not understand them, or because they are more concerned about having new application functionalities than about security [71]. Moreover, also users that are more responsible result to be concerned about the overhead introduced by security mechanism, which may affect performances or the user experience (e.g. interrupting normal activities to send security warnings). Thus, to pursue user acceptance, a security framework should be lightweight, with limited impact on the performance and whose action is as much as possible transparent to the user. Such a framework should show security warning without interrupting the user activity and presenting information accurate, still clear and easy to understand also to non-expert users.

[1] http://androidforums.com/threads/bricked-after-factory-reset.760103/

Wrapping it up, the arguments which motivate and drive our study, i.e. the design of a new security framework for Android mobile devices, can be summarized as follows:

- The problem of malicious applications on new generation mobile devices is of capital importance and concerns billions of users around the world.
- Android is the most popular operative system and it is target of 98% of attacks toward mobile devices.
- Misbehavior of malicious applications seriously affect user's privacy, money and device integrity.
- Identifying misbehaviors of malicious apps is a challenging task and only partially addressed from current security solutions.
- Users hardly accept the overhead and tend to ignore warnings of current security solutions.

1.2 Contributions

This thesis presents the study, design and realization of a security framework, aimed at tackling at different levels the various types of misbehaviors performed by malicious Android applications. This framework combines the action of components both off-line, i.e. static components not present on the protected device and on-line, i.e. on host mechanisms for dynamic monitor and security enforcement. This framework at first attempts to prevent the installation of malicious apps, then monitors the behavior of the device at application, API and operative system level, looking for malicious behavioral pattern which violate system or user-provided security policies. Thanks to monitoring based on hooking, such misbehaviors are blocked as soon as they are detected and before they can be effective. Afterward the application deemed as responsible for the detected misbehavior is safely removed from the device and the details of the malicious applications are shared with the community of users of the proposed framework. The framework has been designed to be lightweight and usable, focusing the runtime analysis only on these apps deemed as suspicious by static components. Moreover, the framework is designed to be as much as possible transparent to the user, requiring an active interaction only to ask confirmation before removal of an application deemed as malicious. Furthermore, the framework has been designed to convey information to the user in a simple but effective way. Tests on user acceptance, presented in the following, confirm that the security information provided by the proposed framework is more effective than the native Android security mechanisms.

The framework also implements a collaborative scheme, which allow framework users to cooperate, sharing knowledge on application behavior and security alerts.

The proposed framework strongly relays on the concept of “contract-based security”, which allows to formally prove the compliance between a security policy and a contract describing the application behavior. To this end, a part of this thesis implement on the Android OS a well established contract-based security mechanism, the *Security-By-Contract* (SxC). A consistent contribution of this work is the extension of the SxC framework with the introduction of *Probabilistic Contract* and *Probabilistic Policies*. The theoretical and practical results of this extension will be thoroughly analyzed in the following.

1.2.1 Framework Architecture

To better describe the contribution, an high level description of the framework is now presented. Figures 1.4 shows the components and the work-flow of the presented security framework. As shown, the framework takes as input an Android

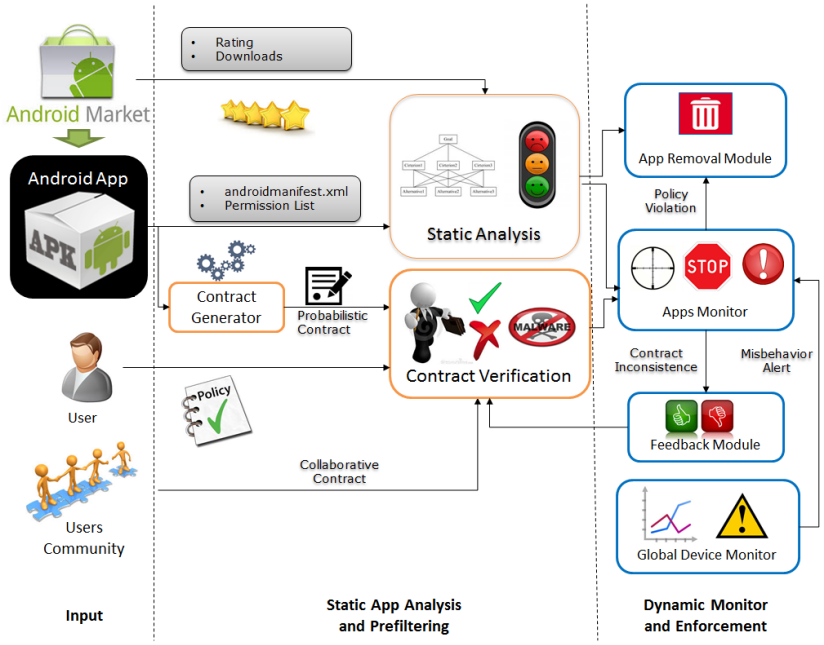


Figure 1.4: High Level Description of the Proposed Framework.

app installation file (`apk` o a new app to install on the device) and a set of information received from the market, the device user and the community of users of the proposed framework. This information is exploited at first by three static analysis components, which verify app security properties and assess the risk before the app is run on the device. Afterwards, four additional components monitor both the app and the device behavior enforcing security policies to prevent misbehavior and to remove malicious apps. A brief description of the main framework components will now follow.

Static Analysis:

The `apk` is actually an archive containing binary files, static resources and `xml` documents which describe the application structure. In particular, the file AndroidManifest.xml included in every `apk` contains the description of the app components and the declared permissions, i.e. the list of resources accessed and critical operations performed. This set of information, together with the data extracted from the market are used by the static analysis component of the proposed framework to evaluate the security risk brought by the application, exploiting the Analytic Hierarchy Process (AHP), a multi-criteria decision algorithm. The analysis is performed at deploy-time, i.e. before the app execution, and the analysis result is presented to the user through a simple and easy to understand index (a colored smiley), which should correctly drive the user final decision on app installation. Details on this components are reported in Chapter 2. For further insight, the interested reader can also refer to [56].

Contract Generator:

The contract generator is an off-line component used to generate an application probabilistic contract. The *probabilistic contract* is a document which extends the AndroidManifest.xml file, presenting a probabilistic description of the application behavior. More specifically, the probabilistic contract specifies the security relevant actions performed by the app, specifying also sequence of actions with their probability to be executed. The contract generally is provided by the app developer. If a developer do not provide the application contract, the contract generator will automatically generate a new one, running the app in a protected environment, attempting to reproduce the app possible behaviors. The contract generator software is very flexible, in fact it can also be used by the app developer itself to build the contract, or in alternative, can be used to monitor and record the behavior of the app while is running on the user device. The proposed framework is then able

to merge the collected executions which may also come by a community of users, to build a still more precise probabilistic contract.

Contract Verification:

The contract verification module is a component whose contribution is two-fold.

1. Verifies if the app probabilistic contract matches a user-defined security *policy*. The matching is verified through probabilistic model checking techniques, before the app is installed on the device.
2. Checks if the app is repackaged (trojanized) exploiting both collaborative probabilistic contract provided by the user community and the one generated by the contract generator.

If the contract matches the user security policy, the app is considered safe and can run on the device without further security checks. If the app is recognized as repackaged and/or the contract does not match the policy, the framework will propose to the user the app removal.

The contract verification and contract generator component will be analyzed in detail in Chapter 3 and Chapter 4. The interested reader can refer to [19].

Apps Monitor, App Removal and Feedback Module:

The apps monitor is a dynamic component of the proposed framework whose task is to analyze the runtime behavior of a subset of applications deemed as suspicious by the Static Analysis or the Contract Verification component. The apps monitor hijacks the security critical operations performed by the app, verifying for each action if:

- The action is part of a behavior allowed by the policy.
- The action is not a known blacklisted misbehavior (heuristics).

If at least one of the two former checks fails (Policy Violation in Figure 1.4), the action is blocked at the act of invocation. Misbehaving apps are reported to the user who can take decisions on the blocked operation, and on the definitive removal of the app. The apps monitor also verifies, for all applications, if their behavior is really consistent with the one declared in the app contract. If a contract inconsistence (Figure 1.4) is found, the contract may be updated and the decision of the Contract Verification module may change accordingly. A *Feedback Module* will handle this contract inconsistence, implementing configurable policies which may include modification to the developer reputation, or communicating the inconsistence to users in the community.

Global Device Monitor:

The global device monitor monitors device events at both API and operative system level. The global monitor is dual and complementary to the Apps Monitor, being able to find a subset of misbehavior that deceive per app monitoring tools. In particular, several operations are not performed directly by the app, instead is the app that demands to the operative system to perform them. In this set of operations fall both the “buffer overflow attacks”, used to illegally get super user privileges and the “confused deputy attacks”, where two or more apps join or re-delegate their authorizations to perform complex attacks. These malicious operations cause noticeable changes in the pattern of issued system calls, detected by the global device monitor, which exploits a statistic classifier to find anomalies. The global device monitor implements a white list detection approach to be able to detect zero day security threats. The global device monitor send threats report to the Apps Monitor component, which exploits heuristics to infer the app that is effectively performing the misbehavior.

The global device monitor will be described in details in Chapter 5. For further details, please refer to [60].

1.2.2 Contribution Summary

It is possible to summarize the contribution of this thesis as follows:

- **Security issues on Android**: The thesis will survey the Android native security mechanisms, the current state-of-the-art solutions to improve or supersede the native mechanism weaknesses and a detailed description of the main security threats to Android device.
- **Framework Description**: A detailed description of the study, design and implementation of the proposed Host-based security framework for mobile devices will be presented. In particular the discussion will focus on the following points.
- **Deploy-Time App Analysis**: A methodology based on a multi-criteria decision algorithm to assess the risk brought by an Android app. This methodology is implemented in a component able to analyze new apps at deploy time directly on the device, without causing any runtime overhead and without extracting binaries or execution code.
- **Contract Generation**: Methodology and tool to extract the application behavior, computing automatically a probabilistic contract from it. This component can run on the user device, in a sandbox, or even rely on a community of users that collaborate to create a contract. Contract can be used to verify if an application is repackaged.

- **Introducing probability in SxC, the SxCxP framework**: We will propose the Probabilistic Security by Contract (SxCxP) as an extension of the contract-based security methodology Security by Contract (SxC). SxCxP substitutes the boolean approach of contract-based security, in which any operation can be either “allowed” or “forbidden”, with *quantitative* clauses. Thus, any operation is considered either “likely” or “unlikely”. This refinement allows to write more complex and flexible security policies, and contracts which take in account the real behavior of an application. The theoretical analysis and implementation details on Android are reported.
- **Multi-Level Monitor and Security Enforcement**: Description of a dynamic component able to monitor the events both at kernel and API level, intercepting the events, classifying them as genuine or malicious thanks to computational intelligence. Malicious events are stopped before they can be effectively performed and the malicious application is killed and removed.
- **Real World Tests**: The effectiveness of the framework as a whole, has been tested against more than 12000 Android applications including both malicious and genuine apps.

1.3 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2 we describe MAETROID, the module of the proposed framework which analyzes new apps at deploy time, assessing their risk level. The multi-criteria decision system is presented, together with a classification of Android permissions. The chapter also reports an analysis on 12000 apps and the survey presented to 200 users to estimate their acceptance. Chapter 3 describes PICARD, the module for the generation of probabilistic contracts and detection of repackaged apps. The chapter presents the theoretical basis under the representation and generation of probabilistic contracts extracted dynamically from app execution. The results on repackaged app detection are also reported. In Chapter 4 it is reported the description of the Probabilistic Security By Contract (SxCxP) and its extension which also includes a Trust module (MAETROID, described in Chapter 2) for contract verification (SxCxTxP). The chapter connects to the results of Chapter 3, reporting the theoretical background of probabilistic contract verification based on ε-simulation. Afterward it propose the implementation on Android systems. Chapter 5 presents MADAM, the global device monitor which enforces security at runtime, detecting anomalous behaviors and malicious applications, exploiting proximity based-classifiers and heuristics. MADAM has been tested against two dataset of malicious apps, accounting

to more than 1300 malicious apps. The detection results, an analysis of the false alarm rate, and of the performance and energy overhead is reported. Finally Chapter 6 briefly summarizes the obtained results, proposing directions for future work.

2

Risk Analysis of Android Applications: A Multi-Criteria and Usable Approach

2.1 Introduction

In the last years, smartphones and tablets have replaced legacy GMS/GPRS (2G) mobile phones. According to a recent market analysis [7], in several countries more than 90% of registered mobile devices have a 3G or 4G subscription, leading to more than 2 billions active subscribers. Noticeably, more than 80% of these devices are based on the Android operating system, which is the most popular operating system (OS) for smartphones and tablets [122]. Such a dominance of use makes Android also the almost exclusive target for mobile threats identified in the last years [5]. Currently, 99% of the Android security attacks are brought through infected mobile applications (*apps*) [5].

Currently, apps for mobile devices are distributed through online marketplaces, such as *Google Play* or *App Store*. These marketplaces act as an hub where app developers publish their own products, which can be bought or downloaded for free by users. Usually, official markets charge users for these apps, while several unofficial marketplaces distribute their own apps free of charge. In this last case, trust is at risk, since there is no centralized control, as it happens with official markets, and it may happen that untrusted developers distribute malicious apps. This issue is particularly serious for Android – which represents the reference of our study – as it is both the most popular OS for mobile devices and the system with the greatest share of malware in the last years [5]. In particular, in 2013 Android accounted for 97% of all mobile malware. Moreover, the number of new malware is alarming: on average, more than 160,000 new specimens are reported everyday [9]. Recently, work in [123] pointed out that a quarter of all Google Play free apps are clones, i.e., repackaged apps of popular ones, such as WhatsApp and Angry Birds.

Android security policy it is not based on a strict control of the app distribution method. In fact, apps are published on the official market without strong security checks and Google is not condemning the presence of unofficial markets. This approach favors a greater flexibility and freedom of choice for both developers and users. Differently from Apple, which enforces application security directly on the market, publishing only apps that have passed a long and complex verification process (vetting), Android enforces security directly on the device. Android, in fact, implements the *Permission System* to force app developers to declare the security critical resources that the app can access and the security critical operations that the app can perform. Then, at run-time, the Android OS blocks any undeclared access or illegal operation attempt. The Android permission system has been proved to show flaws and weaknesses (see [71]), mainly for some assumptions on users' expertise in understanding the permission semantics. In fact, before the app is installed, requested permissions are shown to the user as a list. Then, the user has to decide if the app is *trustworthy* or *risky* by only visualizing the permissions list. Hence, if the user decides that some permissions are not justified, the installation procedure is aborted. Unfortunately, several users may not have enough expertise to understand whether an app is malicious or not by reading the permissions list only. Moreover, according to the analysis in [71], a large number of users does not even read permissions and simply installs the app. In this case, the permission system does not help such users in protecting them from malicious apps. The main criticism that has been raised against the permission system is that they are too coarse-grained and difficult to understand, both for users and developers. We argue that an evaluation of the Android apps trustworthiness deserves more representative methods and techniques.

In this paper, we present *MAETROID* (Multi-criteria App Evaluator of TRust in AndrOID), a mechanism to evaluate the trustworthiness of Android apps, i.e. their level of risk in terms of potential security and privacy risks. To this end, MAETROID performs a static analysis of the app by using five different parameters, found in the app itself or retrieved from the marketplace. Then, MAETROID builds a single easy-to-understand trustworthiness decision for the app, which is shown to the user when installing a new app, i.e. at deploy-time. MAETROID is based on a customized instantiation of a well-known multi-criteria decision process, the Analytic Hierarchy Process (AHP) [108]. AHP conveys in a single index an evaluation performed through criteria which are both objective and subjective. MAETROID exploits AHP to analyze apps through both objective criteria, i.e., declared permissions, number of downloads, and market, and subjective ones, i.e., the app rating and the developer reputation. The outcome of MAETROID is an advise suggesting

the user whether the considered app should be installed or not. The approach of MAETROID is scalable, since apps are evaluated at deploy-time, directly on the user device. Thus, despite the huge amount of available Android apps and their possible different provenances, the proposed approach is viable, since (i) it simply requires the users to install the MAETROID app on their devices (ii) apps are evaluated at installation time only.

We have tested the framework by classifying more than 11,000 apps. Furthermore, we have analyzed the user response and acceptance of MAETROID, designing and proposing a survey to a set of about 200 subjects. The survey results have been analyzed to synthesize outcomes on the users perception of mobile security and related threats. In particular, in our user-set, subjects are aware of mobile security threats, since only 10% of the interviewees state that they are not concerned about possible threats at all. However, more than 25% of the subjects gives no importance to the Android security warnings and only 27% of the subjects considers the Android permissions as useful and meaningful. The results of the survey have also confirmed that the MAETROID evaluation is effective in driving the users decision, by avoiding the installation of malicious apps. In particular, 90% of the interviewees changed their mind about installing a malicious app after that MAETROID evaluates the app as dangerous.

Contributions of the Paper

The contributions of this paper are the following:

- we propose an evaluation of the app threat level based on the analysis of the threats represented by each declared permission. We have rated each of the 145 Android permissions with three threat values, which correspond to threats to user's *privacy*, *device*, and *financial*; these values are used to compute the app global threat score, based on the required permissions. The threat score is the first of five criteria exploited by our framework;
- we describe the design of MAETROID, a system for the analysis of Android apps at deploy-time. Whenever a new app is being installed on the user mobile device, MAETROID exploits five criteria to evaluate the app trustworthiness, then returns to the user a simple decision to help the user in deciding whether to install or not the app;
- we have tested MAETROID against a set of more than 11,000 apps, coming either from Google Play, unofficial markets and two important mobile malware database, namely Genome [136] and Contagio[1];

[1] http://contagiominidump.blogspot.it/

- we present the implementation of MAETROID for Android devices (which can be downloaded from: `http://icaremobile.iit.cnr.it/`);
- we discuss the results about users' expectations and MAETROID acceptance, by detailing the results of a survey presented to almost 200 mobile device users. The survey was aimed at evaluating (i) the users understanding and knowledge of security mechanisms already existing on their mobile devices, (ii) the criteria that average users consider representative of the app trustworthiness, and (iii) whether the MAETROID outcome is more effective than the Android permission system in driving the user towards the right decision (whether to install or not install a new app).

In a nutshell, the current work largely extends and improves the description of previous work in [56]. The completely novel contributions are the implementation of MAETROID as an Android app, the classification of 11,000 apps and the study on the user's acceptance and understanding of MAETROID through the survey.

Structure of the Paper

In the next section, we discuss related work on the security of mobile devices. Section 2.3 recalls some notions about Android security mechanisms and provides a brief description of the Analytic Hierarchy Process. Section 2.4 describes the MAETROID approach by discussing in detail the criteria used for assessing the trustworthiness of an app. The current implementation of MAETROID for Android devices is presented in Section 2.5, which also discusses the results of the analysis on the testbed apps. Section 2.6 presents a survey, which was submitted to 189 subjects, to test user's perception on security on mobile devices and users' acceptance of MAETROID approach. In Section 2.7, we discuss the MAETROID framework, by discussing its advantages and limitations, by comparing it with alternative solutions, and finally we analyze in details the results and findings of the survey. Finally, Section 2.8 draws some conclusions and proposes some further future research directions.

2.2 Related Work

Several extensions and improvements to the Android permission system have been recently proposed. The work presented in [126] proposes a security framework that regulates the actions of Android apps defining security rules concerning permissions and sequence of operations. New rules can be added using a specification language. The app code is analyzed at deploy-time to verify whether it is

compliant to the rule, otherwise it is considered as malicious code. With respect to this work, MAETROID does not require the code to be decompiled and analyzed. Indeed, it only requires the permissions list that can be retrieved from the manifest file and other pieces of information that can be retrieved from the website where the app can be downloaded.

Authors of [131] present a finer grained model of the Android permission system. They propose a framework, named *TISSA*, that modifies the Android system to allow the user to choose the permissions she wants to grant to an app and those that have to be denied. Using mocking data, they ensure that an app works correctly even if it is not allowed to access the required information. However, their system focuses on the analysis of privacy threatening permissions and it relies on the user expertise and knowledge. A work similar to TISSA is presented in [92], where the authors design an improved app installer that allows users to define three different policies for each permission: allow, deny, or conditional allow. Conditional allow is used to define a customized policy for a specific permission by means of a policy definition language. However, the responsibility of choosing the right permissions still falls on the user, whilst MAETROID directly shows to the user the risk classification of the app, performing automatically the permissions analysis.

In [61], the authors present a multi-level behavior-based intrusion detection system called MADAM. The proposed system learns the correct devices' behavior and then detects significant deviations signaling an intrusion. The MADAM approach is orthogonal to that of MAETROID because MADAM analyzes the app behavior at run-time, while MAETROID performs a risk analysis before installing the app. In [45], apps have been classified based on their required permissions. Apps have been divided in functional clusters by means of Self Organizing Maps, proving that apps with the same set of permission have similar functionalities. However this work does not differentiate between good and bad (trojanized) apps. Another analysis of Android permissions is presented in [21], where the authors discuss a tool named *Stowaway*, which discovers permission overdeclaration errors in apps. Using this tool, it is possible to analyze the 85% of Android available functions, including the private ones, to obtain a mapping between functions and permissions. This work mainly concerns the analysis of permissions without proposing a direct link between declared permissions and apps security, as with MAETROID. A system to implement security policies on Android devices is presented in [24]. This system is based on the introduction of a monitor of security critical functionalities, which matches the performed actions with security policies defined by the mobile device user. However, the presence of the monitor imposes a *consistent* over-

head. Another security framework based on user defined policies, preventing app to perform non compliant operations, is presented in [59]. The framework attempts to reduce the overhead and to improve the effectiveness through a probabilistic contract based approach. This leads to a probabilistic satisfaction of security requirements.

TrustGo [120] is a framework aimed at classifying mobile apps exploiting a multi-criteria analysis. TrustGo gives users a full description of the security threats brought by an app and also works as an antivirus. TrustGo is catalogue-based: available Android apps are analyzed by security experts and are inserted in a catalogue, checked when a TrustGo user is installing a new app. TrustGo is effective and the catalogue-based approach ensures a good accuracy. However, it is not possible to collect all the existing apps, since only the apps distributed through official channels can be analyzed. Moreover, if an app is updated, it is possible that some security features may change in the new version, i.e., new permissions are added in the manifest and this requires a catalogue update, which may not be triggered in time. On the other hand, MAETROID is independent from the app version, i.e., the app is analyzed "as is". App update will trigger a new classification process. Moreover, MAETROID classifies the app at deploy-time, without requiring any centralized catalogue. Thus, any app can be classified even if coming from unknown marketplaces. Another app classification system is presented in [101], where apps are classified in comparison with formerly analyzed apps. The methodology exploits probabilistic generative models to analyze apps on different criteria including permissions. However, the performed analysis is more effective in creating an awareness on developers in trying to avoid issues like permission overdeclaration, instead of providing an index effective in driving the user decision on the app installation.

Analysis of the Android permission understanding have been performed in [71], where subjects from an university campus have been asked to fill a survey on Android security and on their current approach to the permission security mechanism. The results of this survey matches with the ones discussed in the first part of our survey used to validate the effectiveness of the MAETROID evaluation. In particular, the percentage of users considering permissions when installing a new app is mostly equivalent. Recently, Android has introduced a service of remote monitoring of installed apps, called *VerifyApps* [80], which acts as a remote antivirus. The visual approach is similar to MAETROID: when an app is considered dangerous, the user is advised about the potential threat and asked if she desires to install the app considered dangerous. However, VerifyApps behaves like an antivirus, by looking directly for known malware signature. On the contrary, the risk

analysis of MAETROID is based on different parameters that do not depend from the app code. A dangerous app can be considered malicious by MAETROID even if it is a brand new app with an unknown signature.

A similar approach to MAETROID is Androlyzer [48], which is a web-based service that gives the user a lot of information about the used API, used libraries, privacy leaks, requested permission, which might be too overwhelming for an ordinary user. For these reasons, in MAETROID we have decided to keep the output of the results as simple as possible as a first step towards a better understanding of the risk of an app from the point of average users. Furthermore, these reputation services, again with similar ones [22] [127], are usually centralized, hence they are not very scalable. In fact, these services need, first of all, to download all the apps (or the most important ones), and are usually limited to unofficial markets. Furthermore, their databases need to be constantly updated and the centralized service need to cope with several concurrent requests of different users. On the contrary, MAETROID is run locally on the user device and only for the newly downloaded app so there are not scalability issues due to checking a large number of apps concurrently.

2.3 Background

This section describes the Android permission system and discusses both its strengths and weaknesses. Then, it recalls the Analytic Hierarchy Process (AHP), the multi-criteria decision system used by MAETROID.

2.3.1 Android Permission System

To reduce the likelihood that a user installs a dangerous apps, Android implements an access control mechanism called *Permission System*. The Permission System forces app developers to declare the security critical resources that the app can access and the security critical operations that the app can perform. At run-time, an Android component called *permission checker* monitors the access requests to security critical resources and operations. If an access request is issued by an app without authorization declared in the Permission System, the permission checker denies the access.

Figure 2.1 reports a time-line of the evolution of Android during 2009-2014, by highlighting the major changes concerning permissions and security. In particular, until 2011 existing malware for Android systems were only confined to research proof-of-concepts. In 2011, Android-specific malware started to spread, by

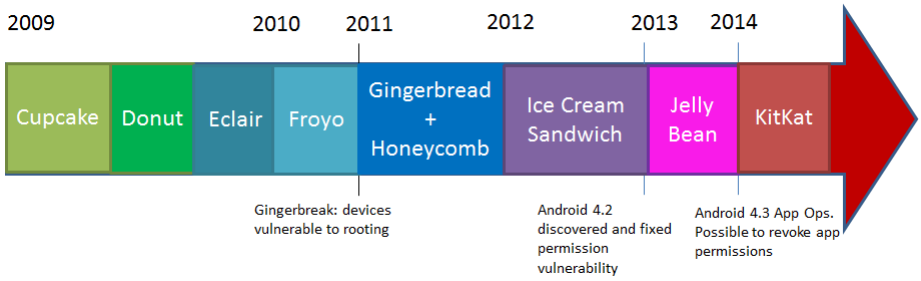


Figure 2.1: Time-line of Major Changes in the Android Permission System.

exploiting a vulnerability of the `Gingerbread` system, named `Gingerbreak`, which allowed malicious apps to get root privileges. This issue has been fixed with the introduction of Android `Ice Cream Sandwich` in 2012. However, in 2013, another important permission vulnerability has been discovered (see [74]) that allowed the modification of the app permissions without modifying the app signature. This issue has been solved in the latest release of `Jelly Bean` (2013), which also includes for the first time the possibility to dynamically revoke, or grant, single permissions to apps.

During the period of observation (2009-2014), starting with the original set of 90 permissions, a further set of almost 50 permissions has been added, mainly due to new device resources and apps functionalities. Currently, Android defines 145 permissions[2], where each permission is related to either a specific device resource or a critical operation. Permissions required by an app are declared by the developer in the `AndroidManifest.xml` file (*manifest* for short), which is included in the app package (`apk`), bound to the app code by means of digital signature. Android classifies permissions in four classes: *normal*, *dangerous*, *signature*, and *signature-or-system*. For the scope of this paper, we only focus on normal and dangerous permissions. In fact, signature and signature-or-system permissions cannot be required by any app. Only apps signed with the Google private key, thus developed by Google, can declare those permissions. The rationale behind signature and signature-or-system permission is that Google is directly interested in providing only genuine apps.

The Android permission classification is used to choose which permissions are shown to the user at deploy-time. All dangerous permissions are automatically shown to the user, whereas the normal ones are listed in a separate sub-list, the

[2] `http://developer.android.com/reference/android/Manifest.permission.html`

"Other Permissions" list. Once a permission has been granted, the app can access the corresponding protected resource (or perform some corresponding critical operations) without asking for further authorizations.

Several criticisms have been raised against the Android permission system. Firstly, the system is considered too coarse-grained [131], since the user can only choose whether to accept all of the permissions declared by an app or to refuse to install the app. Even if the latest versions of Android includes the `AppOps` feature, which allows users to revoke selected permissions to an already installed app, issues related to the coarse granularity of the system still exist. In fact, anytime the app tries to perform an operation for which the permission has been revoked, the operation is denied by the permission checker. Thus, since the error coming from the denied operation is not handled, the app is likely to terminate with error (it crashes).

One of the problem with such an approach is that a user is generally unable to determine if an app can be trusted. In fact, by only looking at the list of the required permissions it can be noticed that the list is not very user-friendly. An example of this is depicted in Figure 2.2, where we can see that it is difficult to fully understand the risk posed by such permissions. Since the number of requested permissions is rather large, and since some of them are quite difficult to understand, even for expert users, several users simply ignore them when installing a new app, leading to malicious apps being installed [71].

Another issue is that often developers declare (by mistake or for convenience) more permissions than those actually necessary, leading to the so called *Permission Overdeclaration* [21]. This happens because some permissions have similar names and their description is not self-explicative for some developers. It is quite intuitive that users, seeing a very long permissions list, are less encouraged to read and understand them.

2.3.2 Analytic Hierarchy Process

The Analytic Hierarchy Process (AHP) [110] is a multi-criteria decision making technique, which has been largely used in several fields of study [111], including, e.g., work in [44], in which AHP has been applied to multi-factor reputation systems, work in [96], [40], and [104], all of them presenting apps related to security policies and access control.

The AHP approach is the following: given a decision problem, where several different *alternatives* can be chosen to reach a *goal*, AHP returns the *most relevant* alternative with respect to a set of previously established *criteria*. The decision

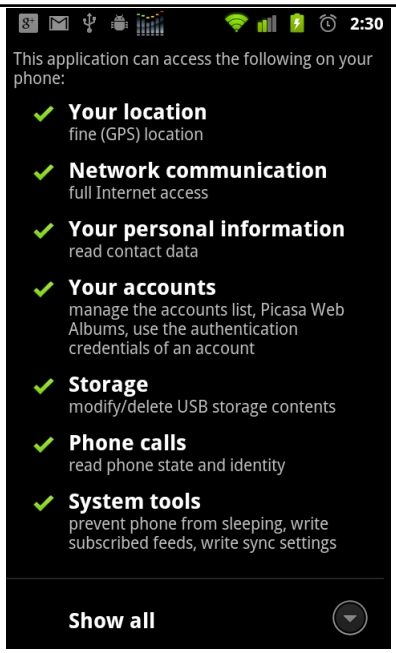


Figure 2.2: Example of Permission List for an App.

problem is structured as a hierarchy, by linking goals and alternatives through the chosen criteria (Figure 2.3). AHP subdivides a complex problem into a set of sub-problems, equal in number to the chosen criteria. For each criteria, a local solution is computed. Then, the most relevant alternative, i.e., the best solution for the decision problem, is computed by properly merging the local solutions.

Differently from classical computational intelligence or statistic techniques used for classification, the AHP decision process does not require a training phase, because decision parameters are assessed directly by experts of the decision problem. The value of these parameters are based on both objective and subjective interpretation of the problem elements. For example, let us consider a decision problem in which one has to decide which car is the best to buy. An example of an objective criterion is the “price”. Instead, if we consider the “aesthetic” criterion, its value is the output of a subjective evaluation. In general, in AHP problems it is up to some experts to assess the relevance of each criterion and the resulting assessment is generally subjective.

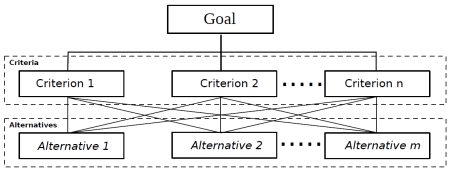


Figure 2.3: Generic AHP Hierarchy

Table 2.1: Fundamental Scale for AHP

Intensity	**Definition**	**Explanation**
1	Equal	Two elements contribute equally to the objective
3	Moderate	One element is slightly more relevant than another
5	Strong	One element is strongly more relevant over another
7	Very strong	One element is very strongly more relevant over another
9	Extreme	One element is extremely more relevant over another

Pairwise Comparison Matrices

Local solutions for each criterion are computed by means of *comparison matrices*, which describe how much an alternative is more or less relevant with respect to another one in a pairwise fashion. For each criterion, the relevance of each alternative with respect to others is expressed in a matricial form. A pairwise comparisons matrix M is a square matrix $n \times n$ (where n is the number of *alternatives*), which has positive entries and it is reciprocal, i.e., for each element a_{ij}, $a_{ij} = \frac{1}{a_{ji}}$, where $a_{ij} \in \{1, ..., 9\}$ (see Table 2.1).

The concept of *consistency* is defined for comparison matrices. A comparison matrix of size $n \times n$ is consistent if $a_{i,j} \cdot a_{j,k} = a_{i,k},\ \forall(i, j, k)$. If a comparison matrix is consistent, the pairwise comparisons are well related between them. However, it is difficult to obtain perfectly consistent matrices using empirically defined comparisons. AHP requires that comparison matrices are, at least, semi-consistent. To measure the consistency of a comparison matrix, the consistency index $CI = \frac{\lambda_{max} - n}{n-1}$ has been defined [109], with λ_{max} being the largest matrix eigenvalue. For a consistent matrix, we have that $CI = 0$, whilst a matrix is considered semi-consistent if $CI < 0.1$. If this condition does not hold, the comparison matrix should be re-evaluated.

Computing Local Priorities

Local priorities express the relevance of the alternatives for a specific criterion. Given a comparison matrix, local priorities are computed as the normalized eigenvector associated with the maximum eigenvalue of the matrix [109]. Thus, for each criterion c_j, AHP extracts from the comparison matrix a vector p_{c_j} of size n expressing the relevance in percentage of each alternative for that criterion.

Computing Global Priorities

The relevance of a criterion with respect to the goal is described by means of an additional pairwise comparisons matrix P_g of size $k \times k$, where k is the number of criteria.

Global priorities are computed through a weighted sum of all the local priorities computed over the whole hierarchy (from alternatives to goal):

$$P_g^{a_i} = \sum_{j=1}^{k} p_g^{c_j} \cdot p_{c_j}^{a_i} \tag{2.1}$$

where $P_g^{a_i}$ is the global priority of the alternative a_i, $p_g^{c_j}$ is the local priority of criterion c_j extracted from P_g with respect to the goal and $p_{c_j}^{a_i}$ is the local priority of alternative a_i with respect to criterion c_j.

2.4 MAETROID

In this section, we describe in detail the MAETROID framework. First, we present a permission classification method, in terms of the amount and the type of the declared permissions to evaluate the potential threat represented by an app. Then, we describe the specific and customized instantiation of AHP to assess the apps' trustworthiness.

2.4.1 Threat Score

Each Android permission regulates the access to a specific resource or operation on an Android device. Generally, the more permissions are declared by an app, the greater is its potential threat, since more security critical operations are granted to the app. However, the number of permissions should not be the only parameter to assess the global app threat level. In fact, some permissions are related to operations or resources much more critical than others. For example, the permission

to send text messages represents a different threat from the permission to control the smartphone vibration. For this reason, we propose a novel way to compute the *threat score*, i.e. the level of risk, of an app based upon the number and typologies of the declared permissions. MAETROID performs a static analysis of declared permission on new apps to compute a threat score representing the dangerousness of the app. This score is represented through a number ranging over the interval $[0, 15]$, where 0 represents an app that requires no permissions, or only harmless permissions, whilst 15 represents the worst case of an app that requires all the Android permissions, or only dangerous one. The value is proportional to the dangerousness of the requested permissions, as will be detailed later.

To properly compute the threat score, we have manually analyzed all the permissions defined by Android. This operation has been done to assess what resources and operations can be accessed through each permission and what is the effect of the misuse of such resource. In particular, each permission has been scored according to the level of threat represented against three security parameters, namely: *privacy* threat (threat to the confidentiality of the user), *system* threat (threat for the system integrity), and *financial* threat (threat for user's mobile credit). Finally, we have created a table which assigns to each permission a score for each of the three threats. The score is in the range $[0, 1]$ according to the six levels of severity defined in Table 2.2.

Table 2.2: Threat Levels

0	No Threat
0.2	Low Threat
0.4	Low-to-Moderate Threat
0.6	Moderate Threat
0.8	Moderate-to-High Threat
1	High Threat

To explain the rationale behind this permissions analysis, let us consider the permission `android.permission.CALL_PHONE` as an example. We have assigned a score of 0.6 to the corresponding privacy threat, since this permission allows an app to perform phone call without the user being aware of it. For example, this permission enables an app to listen and remotely record the user activities and nearby sounds to infer her behavior. We have further assigned a score of 0.2 to the system threat, since the phone drains the battery faster during a call but it is unlikely that an attacker can exploit phone calls to attack the device integrity by

reducing the battery lifetime. Finally, we have assigned a score of 1 to the financial threat, since an app with this permission can call any phone number included premium-rate numbers. Hence, a malicious app can cause a consistent financial damage to the user, e.g. by issuing calls to premium numbers which are likely to pass unnoticed until the user credit ends or the user receives the phone bill.

Table 2.15 in Appendix 2.10 gives an excerpt of the threat score we have assigned to all the Android permissions (for the full list please refer to [55]), where acronyms **PT**, **ST**, and **FT** are an abbreviation of *privacy*, *system*, and *financial* threat, respectively. The threat values have been given according to the documentation associated with each permission, which gives details on how permissions can be exploited by an app. The rationale behind the assignment of threat values to permissions is detailed in the following.

Privacy Threat. According to [119], personal data, i.e., any information referring to an identified, or identifiable, individual (i.e., the *data subject*), are subject to principles aiming to guarantee and preserve their privacy. As an example, the *Use Limitation Principle* states that "*Personal data should not be disclosed, made available or otherwise used for purposes other than those specified at the time of data collection except: i) with the consent of the data subject, or ii) by law.*".

Permissions that refer to actions that may compromise or shatter these principles, such as permissions about access to users' contacts, files, Internet bookmarks and chronology, or SIM and device information, such as the IMEI and IMSI codes, have received a high value for this index. In this cases, such permissions are considered risky from the privacy point of view. In particular, based on the principles highlighted in [119], we have given the highest privacy threat value to the following permissions:

- `android.permission.READ_CONTACTS`: access the contact list on the device. This may create a serious privacy leakage since an app can send the read data to an attacker through Internet or text message.
- `android.permission.READ_PROFILE`: access information of the user account. User's account may contain private data like birth-date, location and occupation and apps with this permission can read, store and eventually send it outside of the user device.
- `android.permission.READ_SMS`: access the text message Inbox. This may constitute a serious privacy leakage, since the app is able to read both the text and the sender number of all private text messages stored in the device.

- android.permission.RECEIVE_SMS: allows the app to control and handle the event of incoming text message. Similar privacy risk of the READ_SMS permission, but the app can only intercept incoming messages and is not able to access the history. However, the app with such a permission may even decide not to show any notification of the received message to the user.
- android.permission.RECEIVE_MMS: allows the app to control and handle the event of incoming multimedia message. Same considerations for the former permission but applied to multimedia messages.

Moreover, a medium-to-low value has been assigned to those permissions that access sensors, such as camera or microphone, since they could be potentially exploited to spy the user behavior. All the remaining permissions, which neither access sensitive resources nor use a resource at all, have been given a low or zero value.

System Threat. A high value of system threat is assigned to permissions accessing system components and that if misused can cause integrity issues to the OS, to personal files, or even the physical device. A list of permissions which are critical on the system threat is the following:

- android.permission.INSTALL_PACKAGES: allows the app to install new packages. This functionality has been used by several malware to install later other dangerous apps with additional permissions (e.g. the ZFT malware) or advertisement apps (Adware).
- android.permission.WRITE_EXTERNAL_STORAGE: allows the app to modify the external memory content. An app with this permission can fill the device memory and remove or permanently modify files of the user of other apps. As an example, the malware Moghava permanently damages all the pictures in the user gallery overlapping on them a propaganda image.
- android.permission.CHANGE_WIFI_STATE: gives to an app the control on the WiFi device status. The WiFi interface has a consistent impact on the device battery lifetime and also may cause Internet disconnection, since WiFi overrides the mobile data connection even if the access point is not connected to the Internet. Thus, a malicious control on the WiFi represents an integrity violation.

Other permissions with a consistent value of system threat are all those permissions that give access to device interface and peripherals (e.g. camera, vibration, etc.) whose (mis)use cause both a performance or battery overhead.

Financial threat. High values of this index are assigned to permissions related to the usage of services that imply a financial cost, such as phone calls or outgo-

ing SMSs. Conversely, if the cost is indirectly related to a specific permission, it receives a medium financial threat value. Some permissions that we consider critical for financial threat are the following:

- `android.permission.SEND_SMS`: allows an app to send text messages. With this permission an app can virtually send as many messages desires to whatever number. Sending text message is an operation which has a monetary cost established by the provider and that may vary with the recipient. Moreover text messages can be used for subscription to premium services which impose a weekly or monthly cost. Such a strategy has been exploited by several malware known as SMS Trojan [130].
- `android.permission.CALL_PHONE`: gives to an app the authorization to initiate phone calls. Phone calls have the same financial implications of text messages, with usually a higher cost which can be imposed on the user [4].
- `android.permission.INTERNET`: gives to an app the authorization to open sockets for external connections. Bytes of data received and transmitted is another element that telephony providers charge to users. Opening a connection and streaming data on it always generates a cost and an app with this permission can virtually send any amount of data. This permission becomes particularly dangerous if coupled with the `CHANGE_WIFI_STATE` permission discussed formerly, whose financial threat is in fact considered moderate (0.6).

An additional detailed example of threat score assignment, which consider the privacy, system and financial threat is given in Table 2.3 for the `SEND_SMS` permission and discussed in the following.

`SEND_SMS` Permission

The `SEND_SMS` permission enables an app to send SMS messages, also without requiring user confirmation. Hence, an app that declares this permission can send SMS messages, with any text, at the current rate, and at any phone number, without the user noting it (unless the user checks periodically the available credit). This permission has been exploited by several malware to leak the user credit by sending messages to premium-rate numbers, or to threaten her privacy by sending information, such as the IMEI and IMSI codes, to a phone number controlled by the attacker [130]. Therefore, we set its financial threat to “High”. The financial threat is considered high since sending SMS text messages has a cost and they can be used to perform subscriptions to premium services. The privacy threat is

considered medium-high since SMS messages can be used as a vector to steal sensitive information. However, this information has to be accessed before it can be sent and this requires other specific permissions. Finally, sending text messages does not represent a threat to the system itself. Hence, the system threat is set to zero. It is worth noting that the threat levels have been assigned only to the permissions defined by Google.

Table 2.3: Threat Level of `SEND_SMS` Permission

Permission	Privacy Threat	System Threat	Financial Threat
`SEND_SMS`	0.8	0	1

Global Threat Score

For each app α, we define the *global threat score* σ, which summarizes in a single index the threats of all the requested permissions declared. This is done by analyzing the manifest file, and by calculating a weighted sum as follows:

$$\sigma = \frac{\sum_{i=1}^{n} w_p \cdot pt_i + w_s \cdot st_i + w_f \cdot ft_i}{1 + \lceil log(1+n) \rceil} \tag{2.2}$$

where n is the number of permissions declared by a specific app, pt_i, st_i, ft_i are, respectively, the privacy, system, and financial threat of the i-th permission required by the app, and w_p, w_s, w_f are the corresponding weights. In the current implementation, we consider w_f being three times greater than w_s and w_p: this means that we consider the financial threat more relevant than the system and privacy threats, since it can harm the user with more impact. The denominator of (2.2) has been added so that the dangerousness of the permission is considered more relevant than the number of permission. We consider apps with σ lower than 4 as *low-threat* apps, while ones with σ in the interval $[1, 4[$ are *moderate threat* to *high-threat*. Higher values of σ mean *extremely* critical apps.

The value σ estimates how much an app is critical from the security point of view, by considering the declared permissions only. Hence, the more permissions are required by an app, and the more dangerous these permissions are, the more critical the app becomes. The idea is that if an app receives a low-threat score,

this should increase the likelihood that this app is downloaded and, as a consequence, this should encourage developers to accurately choose the permissions required by their apps. However, several apps actually require a large number of permissions to perform all their functions, especially communication and social apps, and they should not be considered as suspicious. This leads us to rely on a multi-criteria decision system (presented in Section 2.3.2) in order to classify an app with respect to a set of criteria, among which the threat score σ.

2.4.2 Classification Problem Instantiation

Since the global threat score σ, computed from the requested permissions, should not be the only parameter used to assess the trustworthiness of the app, we instantiate the AHP decision methodology to include further criteria. In detail, an Android app is described by the following parameters: a threat score σ, a developer δ, a number of download η, a market μ, and a user-rating ρ. The *goal* consists in assigning the app one of the following *alternative* labels:

Trusted. The app works correctly and does not hide malicious functionalities. A trusted app is characterized by a low threat score, i.e. it is considered not be able to harm the system due to the low threat of the required permissions. Moreover, a trusted app generally comes from the official market, downloaded by thousands of users, having very good reviews and/or developed by a developer with outstanding reputation (i.e. top developer). All the aforementioned features shape an app which is both secure and appreciated by the users. Therefore, the user can safely install such an app. MAETROID will show a green “happy” smiley when installing a trusted app.

Medium-Risk. The app does not work correctly and includes unwanted functionalities. An app is considered to represent a Medium-Risk to the device security when, even if it shows an acceptable (low) threat score, it has received poor reviews, or has been downloaded by too few users (less than 100) to infer that the app does not hide threats. Generally this decision is given to low quality apps published on official or unofficial market by non-skilled developers. Another reason is that the app is likely to be unwanted from the user, such as an Adware, who should rethink about installing it. A yellow smiley with a neutral expression (“poker face”) is shown to the user when installing a Medium-Risk app.

High-Risk. The app likely includes malicious code. This decision is given to apps that require several dangerous permissions, representing a potential threat to the device and its user. In fact, the greatest majority of malware (95% [5]) asks

for several dangerous permissions related to text messages, which reasonably should not be asked by any app which is not related to instant messaging. As discussed in the following, apps that genuinely ask for several dangerous permissions are recognized by MAETROID thanks to the positive user reviews combined with a conspicuous number of downloads (more than 100,000).

Figure 2.4 gives a graphic description of the AHP instantiation of the MAETROID classification problem. The variables σ, δ, η, μ, ρ represent the problem *criteria*, but, differently from classic AHP instantiation, for each criterion more comparison matrices are defined. A specific comparison matrix is chosen for each criterion, based on its variable value. Hence, for two different apps, the AHP classification problem is instantiated with different comparison matrices[3], representing however the same criteria.

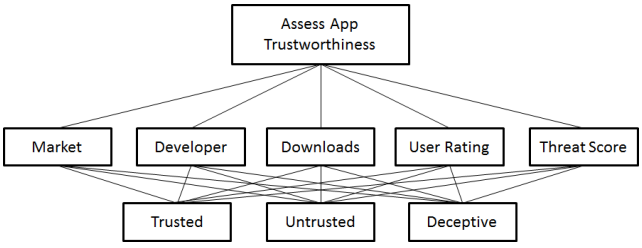


Figure 2.4: AHP Instantiation of the MAETROID Classification Problem.

In the following we detail the building and choice process for AHP comparison matrices used by MAETROID and their relation to the possible values of each of the criteria.

Criteria

We have defined five criteria, namely μ for the market, δ for the developer, ρ for the user rating, η for the number of download, and σ for the threat score. They are detailed in the following.

Market (μ). Apps are normally distributed through online marketplaces. The most popular market is Google Play, also referred as the *official* market. This market is considered as a more protected environment since app developers build

[3] The whole set of comparison matrices is presented in [55].

their reputation on the base of the apps they publish. More specifically, a developer who wants to publish apps on Google Play has to buy a developer account, at the price of 25$. In exchange, the developer receives a private key which can use to digitally sign her apps [107]. If users report an app as malicious, then this app is removed both from the market and remotely from all the devices that have installed it. Moreover, the developer can be tracked and blacklisted. In addition, Google Play includes some reputation indexes that should help users to understand the app quality. These features make the official market a trustworthy place where to download apps. Nevertheless, several malware have also been found in the official market [5] [13] [103].

There also exists a plethora of *unofficial* marketplaces, among which:

- `http://www.appbrain.com`
- `http://www.aptoide.com`
- `http://www.androiddrawer.com`
- `http://androidlife.ru`
- `http://www.anruan.com`
- `http://www.appsapk.com`
- `http://android.pandaapp.com`
- `http://slideme.org`

These marketplaces do not require developer registration, however still attracts several users since these markets usually give access to apps that are not available on the market or distribute free versions of apps that on the Play store . However, unofficial markets often miss reputation indexes and sometimes there is no control on the quality of the apps, so it is easier to download malicious apps. Related to the market source, in the problem instantiation we also consider another parameter, namely app that are *manually installed*, which happens when the user manually installs the app, e.g. when downloading the app `apk` and installing it through a file manager.

With reference to Table 2.1, we show the *relevance* of each alternative, for the three possible values of μ:

- μ = official: we consider that *Trusted* is moderately more relevant than *Medium-Risk* and strongly more relevant than *High-Risk*;
- μ = unofficial: we consider that *High-risk* is moderately more relevant than *Trusted* and slightly more relevant than *Medium-Risk*;
- μ = manually installed: we consider that *High-Risk* is slightly more relevant than *Trusted* and *Medium-Risk* (that are equally relevant).

According to this information, comparison matrices are directly computed.

Developer (δ). We consider three types of developers: *Standard*, *Top*, and *Google*. Google rewards the best app developers with a *Top Developer* badge, which is reported on each app they publish. Hence, these developers should be considered strongly trusted since they produce high-quality apps and should not be interested in lowering their reputation. On Google Play, *Google Inc.* itself is considered a Top Developer; however, we consider Google more trusted than other developers, given the interest that Google has in the well being of Android users.

All the other developers are considered standard and since the Top Developer badge is only used on Google Play, all developers of apps coming from unofficial markets have been labelled standard as well. We make this assumption because the bound between app and developer is not ensured on unofficial markets.

The comparison matrix for the developer parameter is defined according to the following analysis:

- δ = Google: we consider that *Trusted* is extremely more relevant than *Medium-Risk* and *High-Risk* (that are equally relevant);
- δ = Top Developer: we consider that *Trusted* is very strongly more relevant than *High-Risk* and *Medium-Risk* (that are equally relevant);
- δ = Standard: we consider that the three alternatives are equally relevant.

User Rating (ρ). On several markets, users can rate apps and leave a comment, which can be shown to other users. Rating is generally expressed as a number that ranges from 1 to 5 (or it is normalized in this range). We consider apps with a rate less than 2 as *low-quality*, for which the *Medium-Risk* alternative is extremely more relevant than the *Trusted* one. A score higher than 4 means a *high-to-very-high* quality apps for which the *Trusted* alternative is very strongly more relevant than the other two. Intermediate values mean a neutral comparisons matrix.

Number of Downloads (η). Several markets report the number of downloads for each app. As an example, the so-called “killer apps”, i.e., extremely popular apps, have been downloaded from Google Play more than 100 millions of times. These apps should be considered differently from those downloaded a lower number of times, e.g., less than 100 times. In fact, apps with a very high amount of downloads are popular apps already tested by several users and, usually, more trustworthy. Notice that the number of downloads, though independent, is needed to contextualize the User Rating criterion. In fact, a rating of five stars (out of five), given to an app downloaded by a single user, is practically meaningless. Hence, we define 7 intervals in which the value η

may fall. For very high values of η, *Trusted* is extremely relevant. As the value of η decreases, the relevance gradually turns from *Trusted* to *High-Risk*.

Threat Score (σ). For each app, the threat score is computed as explained in Section 2.4.1. We define the following intervals:

- $\sigma < 4$: *trusted* is very strongly more relevant than *High-Risk* and moderately more relevant than *Medium-Risk*;
- $4 \leq \sigma \leq 7$: *High-risk* is very strongly more relevant than the other alternatives (that are equally relevant);
- $\sigma > 7$: *High-Risk* is extremely more relevant than *Trusted*, and *Medium-Risk* is strongly more relevant than *Trusted*.

For marketplaces without download counters and/or rating systems, we define two additional comparison matrices whose elements are all equal to 1. When using these matrices to describe a criterion, all the alternatives have the same relevance for that criterion. Hence, this criterion will not influence the decision. We have defined 20 comparison matrices, but it is possible to increase their number to have finer, or customized, granularity for each criterion. Finally, it is worth noting that the list of proposed criteria is not exhaustive, and the methodology enable the insertion of other rules to evaluate the alternatives. In the current implementation, we consider all the criteria as equally relevant.

The classification process of MAETROID is depicted in Figure 2.5, which shows that apps belong to one of the three classes of the left-hand side of the picture (safe, unwanted behavior, malware) and are classified using one of the three indexes of the right-hand side (Trusted, Medium-Risk, High-Risk). The parameters used to perform the classification are the market reputation, developer reputation, threat score (computed from the permission list), download number and user rating.

2.5 A Prototype Implementation of MAETROID

The MAETROID framework has been implemented as an app for Android devices. The MAETROID app is composed of an activity[4] and several services[5] running in background. Whenever a new app is being installed, MAETROID intercepts the event broadcasted by Android through an intent filter[6]. Hence, the MAETROID app

[4] `http://developer.android.com/reference/android/app/Activity.html`
[5] `http://developer.android.com/guide/components/services.html`
[6] `http://developer.android.com/guide/components/intents-filters.html`

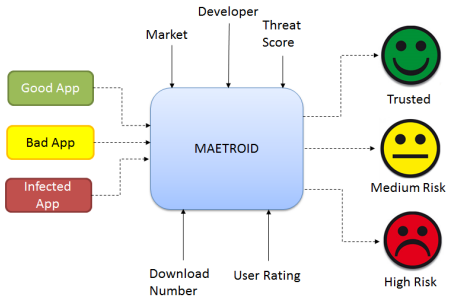


Figure 2.5: MAETROID Classification Process.

comes in foreground with its activity, showing to the user that the app is being analyzed. In the meanwhile, analysis services retrieve the values of the five criteria (market, developer reputation, user rating, number of downloads, threat score) for the app being analyzed. The threat score is computed by parsing the app manifest file, to retrieve the set of requested permissions, and by computing the global threat score as shown in Section 2.4.1. Note that the evaluation is performed locally on user-devices whenever a new app is going to be installed. Hence, there are not scalability issues on the market, since the market is not affected by the computation or by the threat values.

The market is inferred from the installer of the downloaded app. In fact, as discussed in Section 2.4, both official and unofficial markets provide an on-device custom app called “installer”, which is used to browse the marketplace, download, and install some selected apps. The name of the installer is reported in the message which is broadcasted by the OS to communicate the event of a new app installed on the device. The other criteria, namely user rating, number of downloads, and developer reputation are extracted by parsing the HTML code of the market web page. Upon computing the values for the five criteria, MAETROID implements AHP through `Jama`, the Java matrix package for matrix calculi[7].

Upon completing the analysis (left screenshot in Figure 2.6), MAETROID returns its decision in the form of a smiley (Figure 2.6, second, third and fourth screenshots). We have decided to use a simple output decision in the form of a smiley to make it more user-friendly and understandable also by ordinary users. If the app is considered High-Risk or Medium-Risk, the user is advised of the potential threat (through a red ‘sad’ smiley or a yellow ‘poker face’ smiley) and asked if she wants to uninstall such an app. If the user decides to uninstall the app,

[7] `http://math.nist.gov/javanumerics/jama/`

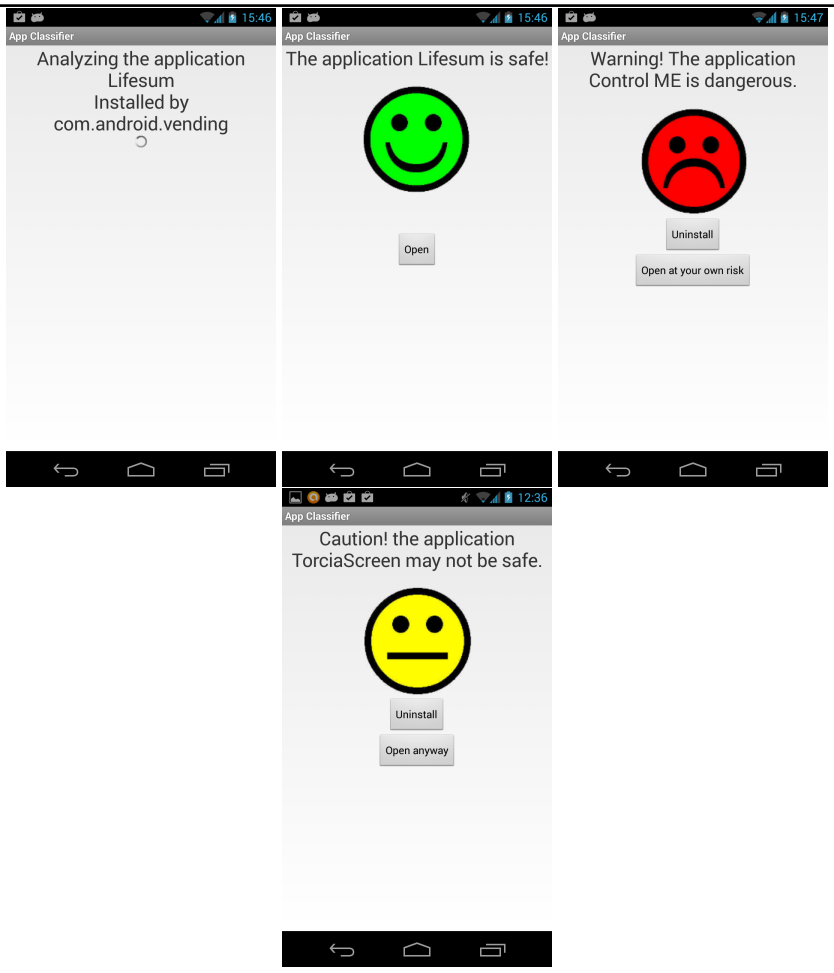


Figure 2.6: Some Screenshots of the MAETROID App.

MAETROID handles the unistallation process. Otherwise, if MAETROID considers the app as trusted, the user is invited (through a green 'happy' smiley) to run the installed app. It is worth noting that MAETROID does not block the installation process, but prevents the app from being started until the analysis outcome is shown and the user has taken her decision. The fact that the installation process is not blocked a priori by MAETROID is not harmful. In fact, in Android an app does not perform any operation (including deploying assets and saving files on the device) until it is run. Given that users follow the MAETROID advise, a dangerous

app can neither harm the user nor the device, since apps can be opened only after that users trigger the app start from the `launcher` app.

It is worth noting that the app classification is performed directly on the device by the MAETROID app. This approach has the following advantages:

- MAETROID is not affected by scalability issues, since it is not necessary to classify “a priori" all the existing Android apps, building a centralized database which would need continuous update and maintenance;
- app updates automatically trigger a reclassification process as soon as the updated version is downloaded. Thus, if the classification result changes because the updated version asks for more permissions, the new version will not be run on the device, given that the user follows the MAETROID decision.

The sequence of steps of the MAETROID analysis process are depicted in Figure 2.7.

Step 1: a new app is downloaded locally from the marketplace;
Step 2: the user decides to perform the app installation;
Step 3: the installation process is hijacked (and paused) by the MAETROID app (which is installed on the user device);
Step 4: MAETROID retrieves the metadata used to perform the classification, locally from the app manifest file and remotely from the marketplace;
Step 5: MAETROID exploits the retrieved metadata to apply the AHP classification locally on the device;
Step 6: the decision is shown to the user, in form of a smiley;
Step 7: the user decides whether to continue the installation or remove the app, based on the output of the classification.

2.5.1 Classification Results

To test the ability of MAETROID in detecting potentially malicious Android apps and classifying the risk of Android apps, we have conducted two set of experiments, one on a large set of applications coming from known databases and one on a smaller set of apps manually analyzed. In the first set we have used the classification algorithm of MAETROID to classify a dataset of 11,046 apps. This dataset is composed of 9,804 apps selected from the official market Google Play, while the remaining 1,242 apps come from the database of known malware Genome[136]. To extract the meta-data of the Google Play apps, we have built a crawler to re-

Figure 2.7: Overview of the Steps in MAETROID Process

trieve the rating, download number and permissions set starting from an existing database [8].

The classification results are shown in Fig. 2.8 and also reported in Table 2.4 for the sake of clarity. As shown, malicious apps of the Genome database have all been classified as risky from MAETROID. More precisely 85% of the malicious apps of Genome have been classified as High-Risk and the remaining 15% as Medium-Risk. None of the apps from Genome have been classified as Trusted. We have used the apps from Google Play as a control set. We can see that the greatest share of Play apps (77,37%) have been classified as Trusted, while 22,4% have been classified as Medium-Risk and only 0,23% with High-Risk. It is worth noting that in these tests there is no knowledge beforehand whether the apps coming from Google Play are really secure or infected by malware, but the classification on the control set is plausible. In fact, the classification results show that more than 75% of the apps from Google Play do not represent a threat to security, while about 22% of the apps show some criticalities, usually due to a low number or download, and only a very small number of apps represent a potential threat to security, mainly due to the set of dangerous permission they ask. Additional details on this set of experiments, i.e. links to the tested apps and their metadata, can be retrieved at `http://www.doc.ic.ac.uk/~dsgandur/maetroid/app_list_final.xlsx`.

[8] `https://github.com/MarcelloLins/GooglePlayAppsCrawler`

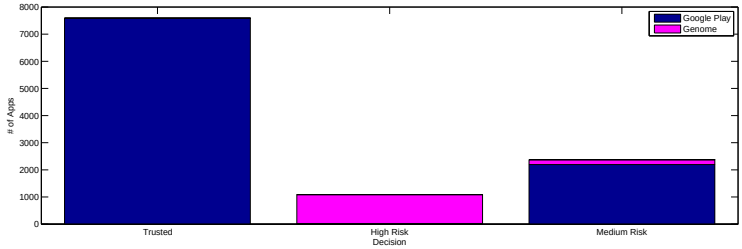


Figure 2.8: Classification Results on 11,046 Apps

Table 2.4: Classification Results on 11,046 Apps.

	Trusted	**High Risk**	**Medium Risk**
Google Play	7586	22	2196
Genome	0	1064	178

In the second set of experiments we have verified the classification accuracy of MAETROID, by measuring both its precision and recall (i.e., the overall classification error). The testbed dataset does not overlap with the previous one and is composed of 180 Android apps, which are known in advance to be:

- *Safe Apps*: those apps that behave correctly both from the security and functional point of view. This class is further divided in two subclasses: *Official* and *Unofficial*, stating, respectively, whether the app has been downloaded from Google Play or not. Good Apps are correctly classified by MAETROID if its output is “trusted" (green “happy” smiley);
- *Apps with Unwanted Behavior*: the app permissions given to these apps may be used to cause potentially unwanted behavior, such as with Adware. These apps are correctly classified by MAETROID if its output is “Medium-Risk" (yellow “poker face” smiley);
- *Malicious Apps*: those apps infected by a malware. Malicious Apps are correctly classified by MAETROID if its output is “High-Risk" (red “sad” smiley).

In more details, the test-set consists of 180 manually vetted apps, of which 90 come from Google Play, 50 from unofficial markets, and 40 are downloaded from Websites that are different from marketplaces (these apps are denoted hereafter with “manually installed"). Among all these 180 apps, 40 are infected by well-known malware. Apps belong to different categories: augmented reality, books and news, communication, desktop manager, entertainment, file managers, game, so-

cial and utility, and anti-virus. The app user rating ranges over $[1, 5]$, the number of downloads ranges over [0, 10M+], and the apps were produced either by standard developers, or by Top Developers, or by Google.

The MAETROID outcome over the test-set is reported in Figure 2.9, where the x-axis shows the three possible outcomes of MAETROID (Trusted, High-Risk, Medium-Risk), while the y-axis shows the number of apps classified per outcome. The light-green color represents safe apps (in dark green the ones coming from unofficial markets), the red color represents apps infected with malware, whilst violet (vertical lines pattern) represents apps with unwanted behavior. All the infected apps have been correctly recognized by AHP as *High-Risk*. It is worth noting that some good apps also fall in this class. These apps come from unofficial markets (labelled as "Good Apps (Unoff.)" in Figure 2.9). Since no user rating is available for these apps, MAETROID applies a safe approach by considering them as High-Risk, at least initially. However, as soon as new information become available for these apps [58], they will eventually be classified as trusted ones. All the apps coming from Google Play have been classified either as *trusted* or *Medium-Risk* based upon the user rating, threat score, and number of downloads. All the apps with unwanted behavior coming from Google Play have been correctly considered as *Medium-Risk*. These apps do not work as expected or they crash upon starting. An example of this class of apps is the game `Avoid the Ghosts`[9], which is a reproduction of the classic Pac Man game. The app does not work correctly: in fact, when the app starts, it is impossible to control the Pac Man movements. The app has been found on the official market, but it has been downloaded few times and it received bad ratings. However, `Avoid the Ghosts` does not require dangerous permissions and, hence, it is considered *Medium-Risk* by MAETROID rather than *High-Risk*.

To better understand the functionalities of MAETROID, in the following we show the classification process for two popular apps, namely Angry Birds Space and Skype.

Classification Example 1: Angry Birds Space

The values of the five criteria in input to MAETROID are shown in Table 2.5. The app developer is a Top Developer. The app has been downloaded by more than 10 millions of users, receiving a global rating of 4.7. Furthermore, it comes from the official market Google Play and it has a low threat score (2.7).

[9] The app was available on Google Play at time of experimentation, while at time of writing it was not available anymore.

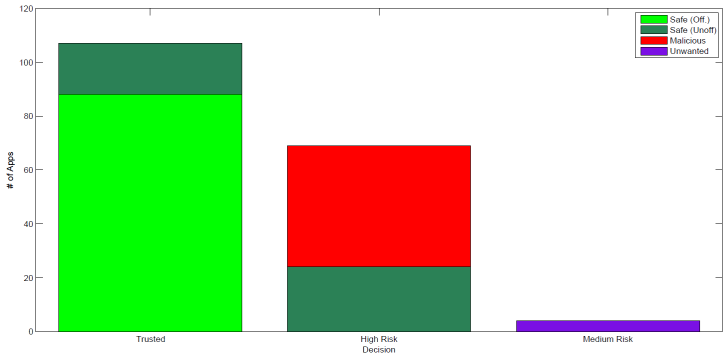


Figure 2.9: Classification Results on Validation Set

Table 2.5: Parameters of Angry Birds Space

σ	ρ	μ	δ	η
2.7	4.7	Google Play	Top Developer	10M+

Table 2.6 shows the matrix used to compare the three alternatives with respect to the “App Developer" criterion for this app. Top Developers generally produce high quality apps and they are not likely to publish malicious apps. Following this intuition, we assigned the following pairwise relevances to the alternatives: *trusted* is very strongly favorite with respect to Medium-Risk and strongly favorite with respect to High-Risk. Trusted (green “happy” smiley) obtains the highest priority (0.7) compared to the other two alternatives.

Table 2.6: Comparison Matrix for Top Developer for `Angry Birds Space`

Top Developer	Trusted	High Risk	Medium Risk	Local Priorities
Trusted	1	4	7	0.7
High-Risk	$\frac{1}{4}$	1	4	0.23
Medium-Risk	$\frac{1}{7}$	$\frac{1}{4}$	1	0.07

Using Equation (2.1), MAETROID merges the local priorities for criterion “developer" with the ones coming from the comparison matrices of the other four criteria, to finally obtain the global priorities $[0.7, 0.16, 0.14]$. The three values rep-

resent the priorities for the three alternatives, respectively *Trusted*, *High-Risk*, and *Medium-Risk*. Trusted is the alternative with the highest value and, thus, it is also the result of the MAETROID classification for that app.

On the contrary, when MAETROID analyzes a version of `Angry Birds Space` found on a database of infected apps it outputs *High-Risk* as the highest priority. This app has been found in the past to be infected by the malware `Geinimi` [130]. The malware steals information concerning both the user and the device, which are sent via SMS to a number controlled by the attacker. To perform these further operations, the malware asks for several other permissions (Figure 2.10), leading to a threat score equal to 7.3. This high value for the threat score correctly drives AHP towards the *High-Risk* outcome.

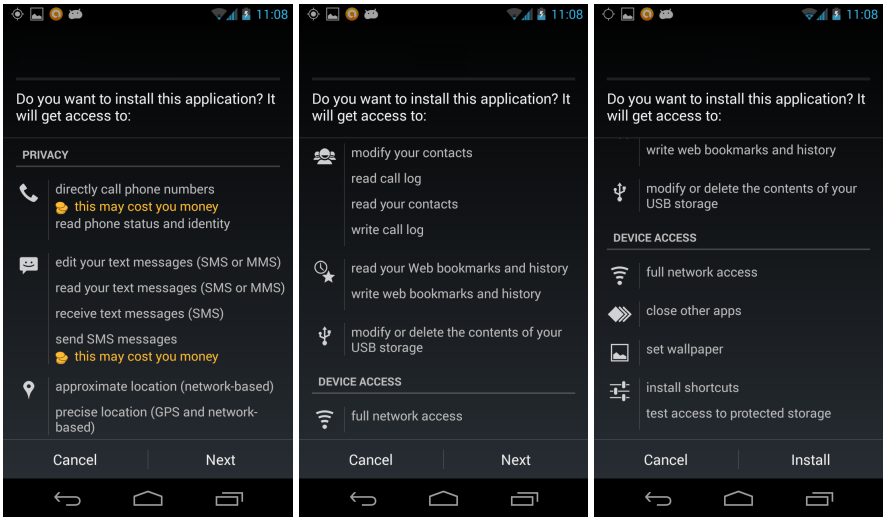


Figure 2.10: Permissions Declared by `Angry Bird Space` Trojanized by Geinimi.

Classification Example 2: Skype

`Skype` is a popular software used for VoIP and free chat and its mobile version has earned a positive outcome, since it enables phone calls with smartphones, using the data connection instead of traditional (and more expensive) landline and cellular calls connections. To work properly, the Android version of `Skype` requires a large number of permissions. Computing the global threat score by means of Equation (2.2), `Skype` gets a score of 6.8. Skype is an example of a high-threat

app. In our analysis, we have considered two Skype versions, one from the official market the other one from an unofficial market, as reported in Table 2.7.

Table 2.7: Two Skype Versions

Name	σ	ρ	μ	δ	η
`Skype 1`	6.8	3.8	Google Play	Standard Developer	10M+
`Skype 2`	6.8	4	Unofficial	Standard Developer	N.A.

MAETROID computes the global priorities vector $[0.47, 0.4, 0.13]$ on the `Skype` version coming from the official market. These results slightly favor the *trusted* alternative (green "happy" smiley). Both the marketplace and the large number of downloads increase the trustworthiness of this app, even if it has a high threat score. For the `Skype` version downloaded from the unofficial market, which does not even provide a download counter, the global priorities are very different from the previous ones: $[0.29, 0.52, 0.19]$ and the app is labeled as *High-Risk* (red "sad" smiley). Even if the two versions require the same set of permissions, it is possible that their source codes are different (possibly malicious). Since more than 10 millions of users have downloaded the version from the official market, it is strongly unlikely that malicious behaviors have not been noticed and reported, forcing the app removal from the market.

2.6 Evaluation of MAETROID Effectiveness: User Expectation and Acceptance

In this section, we discuss the effectiveness of the MAETROID approach in conveying its decision, by analyzing the users' response to the trustworthiness index. In particular, we have created a survey with a list of questions related to the security of Android apps. The survey also contains questions to evaluate the usefulness of MAETROID, i.e., how the MAETROID outcome influences the user's decision (whether to install an app or not). The structure of the survey is available both in 2.9 and online at `http://icaremobile.iit.cnr.it/survey/mobilesecuritysurvey.htm`.

The survey consists of twelve multiple-choice questions which are understandable by average mobile device users. The survey has been made available both

online and physically, at a public event about Internet technology[10]. During the period of observation, in October 2013, we have collected 189 responses, by subjects with different age, background, and technical expertise. The subjects have not been formerly instructed about the survey content.

In the following, we present the survey structure and analyze the answers.

2.6.1 Subjects Set

The first four questions of the survey aim at assessing the variety of the sample. In Table 2.8, 2.9, 2.10, we report, respectively. the distribution of age, background, of the subjects. Most of the subjects, as shown in the Table 2.8, range over 18 to 45 years. One third of the subjects are students, 55% are workers, and 17% answer “other" or nothing.

Table 2.8: Age of Respondants.

Age	% of Respondants
< 18	3.17%
$18 - 25$	23.28%
$26 - 35$	38.90%
$36 - 45$	19.04%
> 45	15.87%
No answer	0.52%

Table 2.9: Gender of Respondants.

Gender	% of Respondants
Female	32.2%
Male	66.6%
No answer	1.05%

We have also asked the users to specify their mobile OS. Figure. 2.11 shows the percentage of market share of mobile OSes among the subjects. As shown, 60% of the subjects use an Android-based mobile device, while the 30% of them

[10] `http://www.internetfestival.it/`

Table 2.10: Occupation of Respondants.

Occupation	% of Respondants
Student	29.10%
Worker	54.50%
Other	15.34%
No answer	0.52%

use iOS (other OSs are used by the remaining subjects). These shares are in line with statistical results by a recent market analysis in [7].

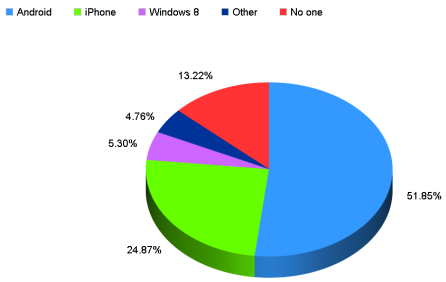


Figure 2.11: Platform Shares

2.6.2 Security Understanding

Questions 5, 6, and 7 are related to subjects' concerns on security threats on mobile devices. In particular, these questions aim at verifying how much the average user is aware of the existence of security threats against their mobile devices, e.g., coming from the installation of non secure apps. Also, the subjects are asked about existing security countermeasures (e.g., anti-virus) and their trust towards them.

Question 5 presents a list of security threats and asks the subject to select those threats that are perceived as the most dangerous. Subjects are allowed to choose from none to all of the presented options. The questions and percentage of subjects that selected the corresponding option is shown in Table 2.11. As shown in the Table, about 50% of the subjects is concerned about the theft of their physical device and attacks to their private data, whilst 40% of the subjects are aware about the possibility of installing malicious software on their mobile devices. These

Table 2.11: Security Concerns of Respondants.

Security Concerns	$\%$ of Respondants
Smartphone theft	50.80%
Identity theft (e.g., Facebook credentials)	33.33%
Credit card data theft	27.51%
Installing Malware (e.g., virus, etc)	35.98%
Attacks to privacy (location tracking)	44.44%
Other	9.52%
None	3.70%

results show that most of the subjects have basic understandings about security problems. It is worth noting that there is a relevant percentage of subjects (about 10%) that are not concerned about security at all. However, according to results, average users are worried about mobile security issues.

Question 6 asks the subjects what security mechanisms and practices they use to protect their mobile devices. Subjects are allowed to select any of the presented options. The questions and percentage of subjects that selected the corresponding option is shown in Table 2.12. As shown in the Table, most of the subjects protect their SIM card through a PIN. A large percentage of users also protect their phone and data through backups and OS update. Several subjects deactivate GPS when not used, but it is possible that most of them are driven by the high battery consumption, rather than by a privacy concern. About one third of users check the app permission list when installing the app. This shows that most of the users (i.e. 2/3) is not concerned at all about the possible risks due to apps asking too many permissions. As we have already discussed, this is because permission list is too difficult to understand for users. Hence, a simple and understandable index as the one proposed by MAETROID is needed. Finally, only a low percentage of subjects (15.87%) has an anti-virus installed on their devices.

Question 7 presents to subjects a list of security features and asks to select those that they would consider useful for their devices. The questions and percentage of subjects that selected the corresponding option is shown Table. 2.13. As shown in the Table, the most required feature is the anti-theft functionality, which is able to lock and/or locate a stolen device. This result is sound with the one of Question 5, stating that users are afraid of having their device physically stolen. Interestingly, the subjects have shown a keen interested in other two security features, namely a classifier of apps (39.15% of preferences), in terms of their hazardous-

Table 2.12: Adopted Security Features and Practices of Respondants.

Adopted Security Feature	% of Respondants
USIM PIN protection	56.08%
Lock screen with password/sequence/PIN	33.33%
Disable GPS when not used	44.33%
Constantly update OS and apps	41.27%
Check permissions required by apps at install time	34.39%
Disable WiFi connection in public places	10.05%
Use anti-virus or secure apps (e.g., to encrypt private data)	15.87%
Backup personal data	31.75%
Other	4.76%
None	10.58%

ness, and the availability of feedback on the app behavior (40.21% of preferences). We want to underline that MAETROID provides both these features.

Table 2.13: Requested Security Features by Respondants.

Requested Security Feature	Respondants (%)
Disable applications	24.33%
SMS blocker	22.22%
Call blocker	20.11%
Anti-theft features (e.g., remotely block a stolen smartphone)	60.85%
Classification of an application hazardousness before installing it	39.15%
Do not install applications from unofficial markets	19.05%
Anti-virus	35.98%
Personal data encryption with a password	32.80%
Send of SMS to secure number in case of anomalous event	40.21%
Block adult content	8.47%
Hide GPS positions to some apps	35%
Secure/anonymous browser	31.75%
Data Backup	33.33%
Disable permissions given to apps	27.51%
Feedback on bad app behavior	40.21%
Other	0.53%
None	5.30%

With question 8, subjects are asked to specify the level of importance that they give to some parameters when evaluating the risk of installing an Android app. In this question, the subject is asked to choose a value of importance, ranging from 1 (very low importance) to 5 (very high importance), for each index. Namely, these indexes are:

- the number of permissions;
- the number of downloads;
- the app rating;
- the app popularity;
- and marketplace.

It is worth noting that these indexes reflect the criteria used in MAETROID to evaluate the trustworthiness of apps. This question has been inserted in the survey to evaluate how much people consider such criteria relevant for defining the app risk.

Figure 2.12 reports the perceived levels of importance for each index. The bars concerning the market index shows that this criterion is perceived as highly important for a large number of subjects (30%). However, almost the same number of subjects (28%) consider the market as a criterion with a very low importance. Quite obviously, the latter are exposed to a high risk of downloading malicious apps. The user rating criterion is considered important by most of the subjects. In fact, only 20% of subjects give this criterion a limited importance (i.e, option “very low” or “low”). On the other hand, only 24% of subjects consider permissions as an important criterion. Instead, 25% of the subjects give no importance to permissions when deciding whether or not to install an app. Popularity and number of downloads are considered from moderately important (medium) to highly important by most of the subjects. Summarizing these results, on average, the importance given to the five criteria is comparable with the relevance we have assigned to the same criteria in the AHP instantiation presented in Section 2.4. The only exception is the *user rating* criterion that is considered slightly more important by the users, and the *permissions* index that is consider less important than the other criteria. Although the user rating criterion is an index perceived as very important by interviewed people, it has two major problems when adopted as the main evaluation criterion. Firstly, its meaning need to be coupled with both the total number of ratings and with the aggregator metric used to compute the global rating [84]. For example, a single, very positive rating, which can also be assigned by the developer himself, does not imply a good quality of the app. A second shortcoming is that, as discussed in Section 2.4, several markets do not include user ratings. These results suggest that users have not a clear understanding of which parame-

ters are more relevant when deciding whether to install or not an app based on the perceived risk. As such, we believe that MAETROID, which incorporates a more careful selection of weights for the indexes, help users in taking a more informed decision.

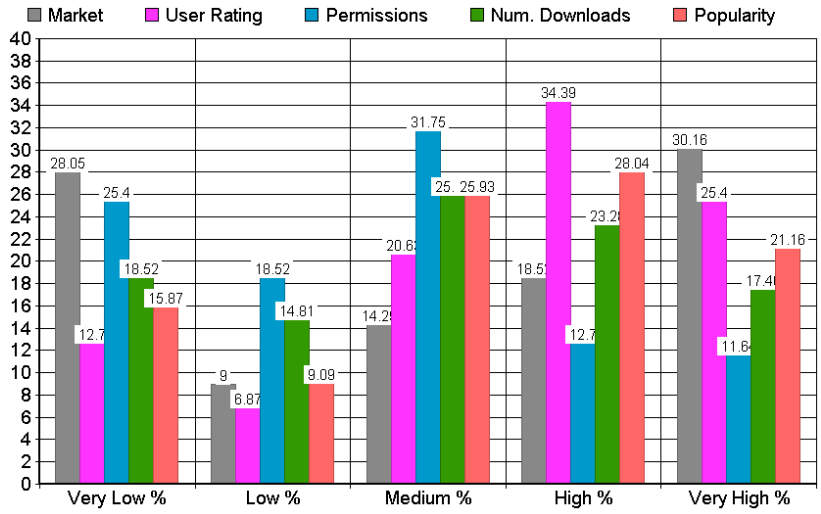


Figure 2.12: Importance of the Evaluation Indexes for the Respondants

2.6.3 Selection of apps

In the last two questions, the survey shows a list of apps with different features, asking which apps the subjects would like to install on their (brand-new) mobile devices. For each app, the survey includes a screenshot of the market that shows the set of requested permission and available criteria (the market name, the developer, and the user rating, when applicable.). The apps included in the survey provide basic features to a mobile device. Each app has three further parameters:

- very popular or unpopular,
- free or premium versions,
- from official or unofficial markets.

The rationale of this question is to understand whether users prefer a free, even if unknown and, hence, potentially harmful app, versus a popular but not free one. The two apps `Angry Birds Space Premium` `Unofficial Free` and

`Ruzzle Unofficial Free` are the trojanized versions of two popular games. These are infected by the trojan GEINIMI, which exploits text messages to register to premium services.

The question has two variants: the first one (question 11), without an index helping the users to make an informed decision, while the second one (question 12) also shows the MAETROID classification result represented by a coloured smiley (see Sect. 2.5). As shown in the results of Table 2.14, a consistent percentage of subjects choose the free (and Trojanized) version of the games, instead of the versions coming from the official market. This happens even if the permission list of such trojanized versions declares some anomalous permissions, like `SEND_SMS`, and a large subset of the interviewed users would install such bad apps (probably, this happens just because they are free). Comparing the second column of Table 2.14 with the first one, we notice that now less subjects are willing to install the trojanized apps, after knowing the MAETROID decision (10%). In fact, 90% of the subjects prefer not to install the trojanized apps after having seen the MAETROID classification result. It is worth also noting that users prefer not to install that app even from the official markets.

Table 2.14: Subjects Willing to Install Apps (Questions 11 and 12).

App	% of Respondants Before MAETROID Decision	% of Respondants After MAETROID Decision
Whatsapp (Play) Free	77.25%	80.42%
Skype Whatsapp (Play) Free	77.7%	57.67%
Angry Birds Space Premium (Play) 0,89 €	11.11%	11.11%
Angry Birds Space Premium (Unofficial) Free	14.29%	4.23%
Monkey Jump 2 (Unofficial) Free	1.06%	1.06%
Viber (Play) Free	39.68%	40.74%
WeChat (Play) Free	11.11%	6.35%
Candy Zuma (Unofficial) Free	4.76%	0.53%
Ruzzle (Play) 2,50 €	12.17%	13.22%
Ruzzle (Unofficial) Free	17.99%	5.82%

2.7 Discussion

MAETROID is an app that helps users understand the risk level of downloaded apps, i.e. if they have potential security and privacy risks. Several factors contribute to make an app likely dangerous. The most relevant is the requested set of permissions, which effectively gives the app the "power" to invoke critical functions and access critical resources. This, *per se*, cannot be the only parameter used to decide whether an app is risky or not. In fact, genuine apps may request these permissions to legally access, for example, the contact detail (as a contact manager), or to send SMSs (as a customized text manager), and so on. Obviously, if these permissions were dangerous regardless of the apps, then Android would remove the possibility of using them. Hence, other criteria are relevant, such as the marketplace (an official market has usually a pre-filtering step to remove some malicious or repackaged apps), the developers (the most important ones do not want to risk their reputation), the user ratings and the number of downloads (which are an indicator of the app "goodness").

MAETROID is shipped as a custom app to run on the user-device. Whenever a new app is downloaded, the installation process is paused to run the classification process locally. Then, MAETROID gives the user a user-friendly decision about the app risk level. Then, it is up to the user to decide whether the installation needs to be resumed. We have to point out that MAETROID is not an anti-virus nor an intrusion detection system. MAETROID focuses on elements which are visible to every users, i.e. permissions, market and app popularity, by evaluating them and then generating a single decision, easy to understand. Furthermore, it does not analyze the app's code. On the contrary, anti-virus solutions base their decision by looking statically for known bad signatures inside the app's code (black-list approach). Other approaches, such as anomaly intrusion detection systems, look for anomaly patterns at run-time. For these reasons, we see MAETROID as an orthogonal approach to these solutions. In particular, MAETROID can be the first line of defense in deciding whether an app may be potentially dangerous. As an example, if an app is classified as Medium-Risk (yellow face), one can decide to run it in a sand-boxed environment under control of an IDS.

The analysis of MAETROID, as already discussed in the former sections, is performed at deploy time and no further checks are performed after the user accepts the decision. In particular, MAETROID does not enforce security at run-time, to not impose any overhead on the apps execution. Thus, MAETROID strongly differs, by design choice, from security frameworks like MOCANA [16] or Samsung Knox [39] which are designed for "high-security government or military deploy-

ment", enforcing security from the hardware to app level through trusted storage for remote attestation and a dedicated market where only vetted apps are published. Furthermore, the configuration of these systems are usually centralized and implemented by expert administrators. The MAETROID solution is, on the other hand, designed for a wider set of users, not requiring dedicated hardware nor customized OS, and with little (or none) knowledge of security.

As discussed in Section 2.4, in our analysis we first assigned to all the five criteria the same priority (0.2). After elaborating the answers to question 8 of the survey, we have re-classified all the 180 apps discussed in Section 2.4, assigning different priorities to each of the criteria, namely by ranking them based on the user's perception of importance. More precisely, we considered the "user rating" criterion two times more important than the other criteria and we reduced the importance of the "permission" criterion by half of its value. The new classification of the apps yielded slightly worse results than the one obtained in our first analysis, i.e. with apps more app being misclassified. This is due to the fact that by reducing the importance of permissions there are more chances of installing malicious apps. MAETROID coalesces in a single and easy-to-understand decision index five criteria that describes the security and quality of an Android app. This trustworthiness index could be significant for yielding better advices to the users than the permission system currently adopted by Android devices, as the first results of our survey-based investigation indicate. In fact, as shown by the comparison of the answers to questions 11 and 12, several subjects changed their mind when installing malicious apps after using MAETROID. Those apps that have been classified by MAETROID as highly-dangerous have been selected by less than the 0.5% of the subjects. The number of users willing to install dangerous apps has noticeably decreased after the MAETROID classification results, in particular by 66% for `Angry Birds Space Premium Unofficial` and `Ruzzle Unofficial`, and by 90% for `Candy Zuma Unofficial`, respectively. For apps classified as Medium-Risk, such as `Skype`, a significant percentage subjects are not willing to install the app anymore (25% less than before). These results are encouraging, since they indicate that MAETROID is effective in driving the correct decision when installing a new app.

One limitation of the MAETROID approach is that it uses a single index for evaluation, with three possible values only, and this might somehow be too coarse-grained in some situations. As an example, the vast majority of apps found on official market, such as Google Play, are classified as trusted. This is because it rarely happens that malicious apps are hosted on official markets. However, one could argue that even if safe, these apps sometime requires a too large set of

permissions and should be penalized. Further, advanced users would like to know more about the threats for each of the MAETROID categories (financial, privacy and system), which is not represented by the final output. The first issue can be easily taken into account by penalizing apps that requires a large number of permissions regardless of the market, developer or the number of download. As we have already detailed, the parameters, such as the weight given to each criteria, can be customized by the users. Regarding the second issue, one future development concerns the addition of some further explanations to the final decision, e.g. in the form of an optional tab that the user can open to understand in detail the final decision.

Concerning the results of the classification, we have to say that weights (i.e., the importance) of the parameters can be customized as to have more granular results with respect to the user's expectation. As an example, one can decide to give more importance to the developer rather than to the market, if the user always downloads from official markets. Further, if a particular user is really concerned about the privacy, the matrix weights can be biased towards giving a higher threat score whenever an app includes more privacy-risky permissions. To summarize the results of the survey, we can conclude that subjects are aware of the security threats brought by malicious mobile apps. However, the current Android alerting system, which consists of showing to the user the list of permissions requested by an app before installing it, seems to fail to be effective. As highlighted by the answers to the survey, several subjects gave to permissions a limited importance and they base their decision (either to install an app or not) on non security-related criteria, such as the number of downloads and the user rating, which we have shown no to be relevant.

2.8 Conclusions

Protecting users from dangerous apps is a compelling issue. Though the main mobile OSes have already introduced some security mechanisms for device and user protection, they still present several usability-related flaws. The results of the survey we have conducted show that users have a realistic view of mobile security threats and are willing to protect their devices. However, users are often tricked in installing malicious apps looking as genuine, since users seem to consider the app popularity and user rating more important than, e.g., the declared permissions. To this end we have developed MAETROID, a multi-criteria decision framework for the analysis of Android apps. MAETROID has been exploited to classify more than 11,000 Android apps, coming from Google Play and from a database of known

malware. In our experiments, the trustworthiness index of MAETROID has proved to be able to drive correct decisions on whether to install dangerous apps.

We believe that the introduction of a simple index, as the one produced by MAETROID, may improve the overall mobile device security and user awareness. In fact, suspicious apps could be identified and further analyzed, before being executed by users. Moreover, the presence of the threat score could be an incentive for developers to accurately choose the permissions needed by their apps, effectively tackling also the permission overdeclaration issue. MAETROID comes as an Android app which enforces security without imposing overhead to the user, since it becomes active only when installing a new app.

2.9 Survey Structure

Survey on security for mobile devices. Welcome to the survey on security for mobile devices!

This survery helps users understanding their awaireness of how to securely use their smartphones.

There are 12 questions in this survey.

1. Age of user.

   Please choose **only one** of the following:
   - < 18
   - $18 - 25$
   - $26 - 35$
   - $36 - 45$
   - > 45
2. Gender.

   Please choose **only one** of the following:
   - Female
   - Male
3. Occupation status.

   Please choose **only one** of the following:
   - Student

- Worker
- Other

4. What kind of smartphone do you use?

   Please choose **all** that apply:
   - Android
   - iPhone
   - 8 Windows 8
   - No one
   - Other

5. What are the threats for smartphones you fear most?

   Please choose **all** that apply:
   - Theft of Smartphone
   - Identity Theft (eg., Facebook credentials saved on the Smarpthone)
   - Theft of credit card data
   - Anonymous calls / stalking
   - Installing malware (e.g virus, etc)
   - Attacks to user privacy (access to personal images from unauthorized people, position tracking)
   - None
   - Other

6. Which of the following basic security functionalities do you use to protect your Smartphone?

   Please choose **all** that apply:
   - PIN to protect calling card
   - Lock screen with password/sequence/PIN
   - Disable GPS when not used
   - Constant update of operating system and apps
   - Check permissions required by apps at install time
   - Not use wireless in public places
   - Use of Anti-virus or secure apps (e.g., to encrypt private data)
   - Backup of personal data
   - None
   - Other

7. Which of the following features would you like to see in a security app?

   Please choose **all** that apply:
   - Disable applications
   - SMS blocker
   - Call blocker
   - Anti-theft features (e.g, remotely block a stolen smartphone)
   - Classification of an application hazardousness before installing it
   - Do not install applications from unofficial markets
   - Anti-virus
   - Personal data encryption with a password
   - Send of SMS to secure number in case of anomalous events (e.g change of SIM, settings)
   - Block adult contents
   - Hide GPS positions to some apps
   - Secure/anonymous browser
   - Data Backup
   - Disable permissions given to apps
   - Feedback on bad application behavior (leaking user money, battery depletion, download of apps in background, etc)
   - None
   - Other
8. How much each of the following item is critical when deciding whether to install or not an application (1 little, 5 very much) ?

   Please choose the appropriate response for each item:
   - Market: 1 2 3 4 5
   - User ratings: 1 2 3 4 5
   - Few permissions: 1 2 3 4 5
   - Number of download: 1 2 3 4 5
   - Popular application: 1 2 3 4 5
9. Would you use an application that, automatically, rates the degree of hazardousness of an application you are going to install?

   Please choose **only one** of the following:
   - No
   - Yes, with a score between 1 (harmless) and 10 (dangerous)

- Yes, with a traffic light: red (harmless) / yellow (suspicious) / red (dangerous)
- Yes, with a smiley: smile (harmless) / “poker face” (suspicious) / sad (dangerous)
- Other

10. The score on WhatsApp degree of hazardousness downloaded fron an UNOFFICIAL market shows a red light (sad smiley): what are you going to do?

    Please choose **all** that apply:
    - I’m not going to install it and I quit
    - I’m going to search it in another market
    - I’m going to install it because it’s very popular
    - Other

11. Which of the following applications would you install on a brand new Smartphone?

    Please choose **all** that apply:
    - Whatsapp (Google Play) Free
    - Skype (Google Play) Free
    - Angry Birds Space Premium (Google Play) 0,89€
    - Angry Birds Space Premium (Unofficial) Free
    - Monkey Jump 2 (Unofficial) Free
    - Viber (Google Play) Free
    - WeChat (Google Play) Free
    - Candy Zuma (Unofficial) Free
    - Ruzzle (Google Play) 2,50€
    - Ruzzle (Unofficial) Free

12. If you also had a score on the hazardousness of an application using a smiley, which of the following applications would you install on a brand new Smartphone?

    Please choose **all** that apply:
    - Whatsapp (Google Play) Free
    - Skype (Google Play) Free
    - Angry Birds Space Premium (Google Play) 0,89€
    - Angry Birds Space Premium (Unofficial) Free
    - Monkey Jump 2 (Unofficial) Free
    - Viber (Google Play) Free

- WeChat (Google Play) Free 😐
- Candy Zuma (Unofficial) Free ☹
- Ruzzle (Google Play) 2,50€☺
- Ruzzle (Unofficial) Free ☹

2.10 Excerpt of Analyzed Android Permissions

Table 2.15: Partial list of Android Permissions and Associated Threat Levels, per Index

Permission	Class	PT	ST	FT
android.permission.ACCESS_COARSE_LOCATION	Dangerous	0.4	0	0
android.permission.ACCESS_FINE_LOCATION	Dangerous	0.8	0	0
ACCESS_LOCATION_EXTRA_COMMANDS	Normal	0.2	0	0
android.permission.ACCESS_MOCK_LOCATION	Normal	0	0.4	0
android.permission.ACCESS_NETWORK_STATE	Normal	0.2	0	0.4
android.permission.ACCESS_WIFI_STATE	Normal	0	0	0.4
android.permission.AUTHENTICATE_ACCOUNTS	Dangerous	0.6	0	0
android.permission.BATTERY_STATS	Normal	0	0.2	0
android.permission.BLUETOOTH	Dangerous	0.6	0.2	0
android.permission.BLUETOOTH_ADMIN	Dangerous	0.8	0.6	0
android.permission.BROADCAST_STICKY	Normal	0	0.2	0
android.permission.CALL_PHONE	Dangerous	0.6	0.2	1
android.permission.CAMERA	Dangerous	0.8	0.6	0
android.permission.CHANGE_CONFIGURATION	Dangerous	0	0.4	0
android.permission.CHANGE_NETWORK_STATE	Dangerous	0.2	0.6	0.6
android.permission.CHANGE_WIFI_MULTICAST_STATE	Dangerous	0	0.2	0.2
android.permission.CHANGE_WIFI_STATE	Dangerous	0	0.6	0.6
android.permission.CLEAR_APP_CACHE	Dangerous	0	0.2	0
android.permission.PROCESS_OUTGOING_CALLS	Dangerous	0.8	0.6	0.2
android.permission.READ_CALENDAR	Dangerous	0.8	0	0
android.permission.READ_CONTACTS	Dangerous	1	0	0
android.permission.READ_SMS	Dangerous	1	0	0
android.permission.RECEIVE_BOOT_COMPLETED	Normal	0.2	0.4	0
android.permission.RECEIVE_MMS	Dangerous	1	0	0.8
android.permission.RECEIVE_SMS	Dangerous	1	0	0.8
android.permission.RECEIVE_WAP_PUSH	Dangerous	0.4	0.6	0.6
android.permission.RECORD_AUDIO	Dangerous	0.8	0.6	0
android.permission.REORDER_TASKS	Dangerous	0.4	0.2	0.4
android.permission.RESTART_PACKAGES	Normal	0	0.2	0
android.permission.SEND_SMS	Dangerous	0.8	0.2	1
android.permission.WRITE_CALENDAR	Dangerous	0.8	0.2	0
android.permission.WRITE_CONTACTS	Dangerous	0.6	0.6	0
android.permission.WRITE_EXTERNAL_STORAGE	Dangerous	0.2	0.6	0
android.permission.WRITE_SMS	Dangerous	0.4	0.2	0
android.permission.WRITE_SOCIAL_STREAM	Dangerous	0.6	0	0
android.permission.WRITE_SYNC_SETTINGS	Dangerous	0	0.4	0

3

Detection of Repackaged Mobile Applications through a Collaborative Approach

3.1 Introduction

The large diffusion of smartphones and tablets experienced in the last years has drastically changed the paradigm of application distribution. Currently, applications for mobile devices (apps, for short) are distributed through online marketplaces, such as *Google Play* or *App Store*. These marketplaces act as an hub where app developers publish their own products, which can be bought or downloaded for free by users. Usually, official market charge users for these apps, while several unofficial marketplaces distribute their own apps free of charge. In particular, with unofficial markets users can simply download the app file and install it later without adhering to the official installation steps. In this last case, trust is at risk, since there is no centralized control and it may happen that untrusted developers distribute malicious apps. This issue is particularly serious for Android – which represents the framework of reference for our study – as it is both the most popular operating system for mobile devices and the system with the greatest share of malware in the last years [5]. In particular, Android accounts for 97% of all mobile malware in 2013. The number of new malware is pretty scary: on average, every day more than 160,000 new specimens are reported [9]. Moreover, recently, researchers [123] have found that a quarter of all Google Play free apps are clones, i.e., repackaged apps of popular apps such as WhatsApp and Angry Birds.

The vast majority of mobile malware is Trojan, which comes in the form of *trojanized app* [89]. These apps look like genuine ones, but they run malicious code in the background. Typical misbehaviors of these apps include private data leakage, user position tracking, stealthy outgoing of SMS messages and forced subscription to premium services. Trojan related to SMS and subscription to premium services are extremely common and constitute a Trojan class named *SMS*

Trojan, which accounts for almost 83% of the total Trojan malware. These malicious apps send SMS messages, which by itself is a cost for the user, who will not even notice this misbehavior, since in Android devices SMS sent by apps are not stored in the outbox [54]. Moreover, SMS trojans often send SMS messages used to subscribe the user to premium services. These services charge the user periodically for services, generally multimedia contents, sent via SMS or MMS. However, SMS trojans intercept the action of incoming messages and if they are from a white-list of premium numbers, then they are dropped without notifying the user. Only after a substantial amount of money has been leaked in this way, it is likely that the user notices the misbehavior.

Diffusion of malicious apps is a serious issue. In general, it is well-known that they can be easily distributed through unofficial markets, which are known to be non-secure. However, malicious apps have been also found in the Google Play store. Furthermore, extremely popular apps have been distributed also through uncontrolled channels, like simple Web pages, which thus represent the best target for developers of malicious apps. An example is given by the case of the `Flappy Bird` app. This extremely widespread app has been recently removed from the official market for reasons that are not related to security, but it is still possible to get it through other channels. Currently a considerable share of the `Flappy Bird` app available online are trojanized [11].

Since it is hard for a user to detect misbehaviors due to hidden malicious code, it is worth adopting ad-hoc, automatic, control mechanisms. Android includes security mechanisms to protect the device and its user from malicious apps. In particular, the *Android permission* system provides an access control mechanism for all the resources and critical operations on the device. However, the effectiveness of this system is limited against trojanized apps. The main reason is that permissions are too coarse-grained to express effective security policies [21], while users may find hard to understand the security threat brought by apps by simply reading its permission list [71].

In this paper, which is a revised and extended version of a work presented at CTS 2013 [17], we present PICARD (ProbabIlistic Contracts on AndRoiD), a probabilistic contract-based intrusion detection system to recognize and block the misbehaviors performed by trojanized apps on Android devices. PICARD is a collaborative framework based on probabilistic contracts generated from the execution traces collected by a network of collaborative users. A *contract* is a document that describes the expected behavior of an app. A version α of an app executed by the user is *compliant* with the contract γ related to the app, denoted $\alpha \models \gamma$, when

all the sequences of actions effectively performed by the app are included in the contract.

A contract can be defined using information that can be computed either statically or dynamically. In the static approach, the contract can be built by learning some properties from, e.g., the source code. However, it may be difficult to know in advance some properties of the code, such as the behavior of the app depending on specific inputs or interactions. Moreover, static approaches are based on the availability of the app source code, which is seldom distributed by the app developers. In the dynamic approach, a contract can be defined by exploiting the information learned from app's execution, which is monitored at run-time to extract its behavior. Our methodology focuses on the dynamic approach, as it is more suitable to represent apps whose behavior depends on user inputs. Contracts defined by using dynamically-generated execution traces can also include quantitative information deriving from the direct observation of the app behavior. In particular, such information can be obtained by analyzing the occurrences of any execution trace. As we will see, by enriching the contract with specific information about the frequency with which every behavior is expected to be observed, it is possible to approximate probabilistically the relation $\alpha \models \gamma$. More specifically, since both γ and α include quantitative information, intuitively the idea is to verify whether α is approximately compliant with γ up to some tolerance threshold ξ, written $\alpha \models_{\xi} \gamma$.

Finally, it is worth observing that in the proposed approach the contract is based on execution traces provided by collaborative but possibly untrusted users. Hence, to keep track of the trustworthiness of each user participating in the contract generation, we integrate our framework with a centralized reputation system employed to favor user collaboration and exactness of the contract.

Contributions of the Paper

The main contributions of the paper can be summarized as follows:

- We present PICARD as a collaborative framework in which users share, through a central server, execution traces of the apps running on their mobile devices. This is done through the PICARD app, which is the component running on the mobile device used to collect execution traces of the apps at the system call level and to protect the device itself from app misbehaviors.
- We introduce the concept of ActionNode to describe app behaviors and app contracts through clustered graphs of system calls. Then, based on such a notion, we present (i) an approach to the formal representation of app behaviors and contracts through probabilistic automata, and (ii) a method to collabora-

tively build a contract by merging several execution traces from different users, by taking into account their reputation.

- We describe a method to match the behavior of an app with a probabilistic contract through statistical tests, to discern between genuine and repackaged versions of a mobile application.

Structure of the Paper

The rest of the paper is organized as follows. In Section 3.2, we describe the design of the PICARD framework, by detailing the algorithm used to acquire traces from collaborative users. In Section 3.3, we describe the methodology used to generate probabilistic contracts. In Section 3.4, we define the statistical methods used to match a monitored behavior with the contract. In Section 3.5, we show the viability of the approach through experiments on some trojanized apps. Section 3.6 reports on related work about contract-based approaches and probabilistic techniques to monitor the app behavior. Finally, Section 3.7 concludes the paper by discussing some future works.

3.2 PICARD Framework Description

In this section, we describe in detail the framework of PICARD, which is a distributed, collaborative client-server platform (see Figure 3.1). In this framework, different users share dynamically their experience in the usage of every specific app – in the form of monitored execution traces – through a centralized server. The objective of the server is to employ and combine the behavior of the app, as monitored by every participating user, in order to build incrementally an app contract, which is then broadcast to all the involved users. Basically, no trust assumptions are made *a priori* on users. Hence, to enable a virtuous circle ensuring user collaboration and reliability of the collected information, user reputation is continuously updated and considered during the whole contract lifetime.

As far as the client side is concerned, we assume that each user has a unique identifier assigned by the PICARD server when the PICARD app is installed on the user device. The identifier is based on a fingerprint of user's phone IMEI (International Mobile Equipment Identifier) in order to ensure its uniqueness. Whenever the user downloads a new app, the PICARD app notifies the central server in order to obtain, if available, the app contract, which is signed by the private key (of an asymmetric cryptography key pair) of the server. Then, at the client side the PICARD app implements the algorithm depicted in Figure 3.2. Execution traces

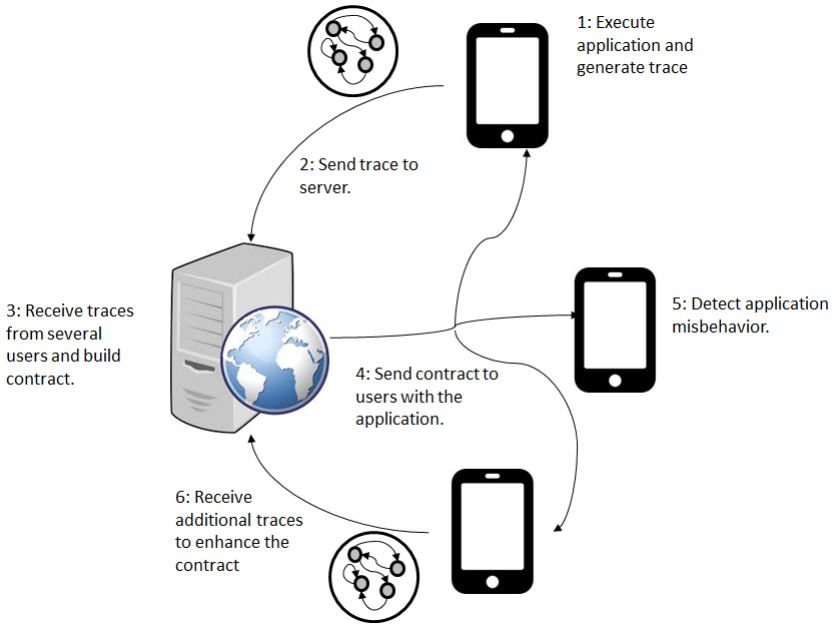


Figure 3.1: PICARD High-Level Architecture

of the monitored app are collected directly and constantly on the user device. If the PICARD app has a contract for the app, then the monitored behavior is used to detect possible misbehaviors with respect to the contract, otherwise it is simply forwarded to the server. If a misbehavior is detected, then PICARD blocks the current action and notifies the user that the app behavior does not match the contract. The user can then choose to stop the misbehaving app, block and uninstall it, or simply do nothing. Notice that if no misbehavior is detected, then the monitored behavior is sent to the server to contribute to the contract maintenance.

PICARD exploits system call analysis to capture and monitor the app behavior at the kernel level. The advantages of performing analysis at this level is that performing a code hijack at such a low level is harder than doing it at an higher level, e.g., API or application level. Hence, the obtained behavior representation should be exhaustive and comprehensive of every action performed by any app. The sequences of system calls resulting from the monitoring activity derive from the app usage during its lifetime. Typically, an execution trace is recorded from the time when the app is started and ends when the app is closed, or after a sufficiently long time of the app execution. When PICARD stops monitoring an app, i.e., the monitored app is terminated or the trace length overcomes a specific threshold,

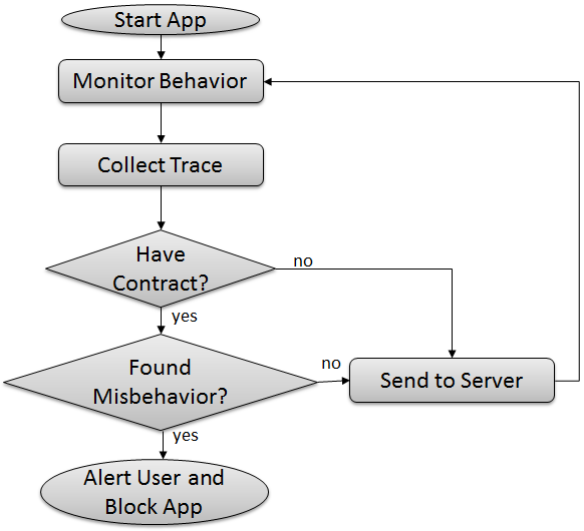


Figure 3.2: Description of the Behavior of the PICARD App

the user is asked to send the execution trace as a report to the PICARD central server. To reduce the required level of interaction, which can be considered annoying by users, it is possible to set the app to automatically send the execution traces when they are available.

On the PICARD server side, a database of users, apps, and contracts is managed. All the operations dedicated to user reputation and contract management are handled by the central server. When a user sends the report concerning an observed execution trace, the PICARD server verifies if the related app is already in the database. If the app is missing, a new record is added. Then, the PICARD server follows the algorithm depicted in Figure 3.3 to manage the app contract on the basis of the received execution traces. As can be noticed, the first operation upon trace reception consists of checking contract's completeness, which is an index describing the reliability of the contract. Intuitively, a contract is not considered complete (and not made available to the users) until enough traces have been received from a sufficiently high number of users. As we will see in Section 3.3, the completeness level is determined through statistical methods contributing to validate the contract.

If the contract is not complete yet and if the reputation of the user exceeds a given threshold θ, i.e., the feedback reported by the user is trusted enough, then the execution trace and the reputation of the user are used to update the partial

contract of the app, as detailed in Section 3.3, otherwise the trace is discarded and does not contribute to the contract. Merging the contributions provided by several different users is fundamental to achieve contract completeness. In fact, if we only consider the traces recorded by a single user (or a small group of users), then the generated contract would be partial since a user rarely explores all of the possible app behaviors.

As soon as the contract is complete, the fitness of any execution trace received is tested against the contract. If the test is passed, then the reputation of the user is increased to reward the collaborative behavior. Moreover, notice that the contract, even if complete, is continuously updated to keep track of additional contributions provided by new traces. If the test is not passed, then it may reveal a potential malicious behavior of the user, whose reputation is decreased. Then, the trace is recorded (and added to the contract) only if the user is trusted enough. In other words, the behavior of users reporting their usage experience is rewarded (or punished) in terms of reputation, provided that such a feedback is (or is not) consistent with respect to the contract. In fact, it is likely that cheating users provide false feedback that, however, does not fit the contract behavior. Hence, the fitness test and the negative reputation variation related to unsuccessful tests represent a way for isolating such behaviors.

The complete version of the contract is sent by the PICARD server to each involved user, while new versions of the contract – resulting from the combination of additional execution traces continuously sent by collaborative users – are released when necessary. These operations are clustered and the updated contract is resent to users periodically. To strengthen the role of reputation as an incentive to promote collaboration, the server may decide to privilege trustworthy users by sending to them the updated versions of the contract more frequently with respect to less trusted users.

Finally, we point out that the dynamic nature of the approach used to define the contract is such that even a complete contract, which is essentially generated through tests, cannot be completely specified in a formal sense. A fully specified contract requires full state space exploration, which, however, could be impractical in real scenarios.

3.3 Contract Generation

In the PICARD framework, generation of an app contract is based on the quantitative analysis of a certain amount of different execution traces that represent the usage of the app by several users. Each execution trace observed by a trusted user

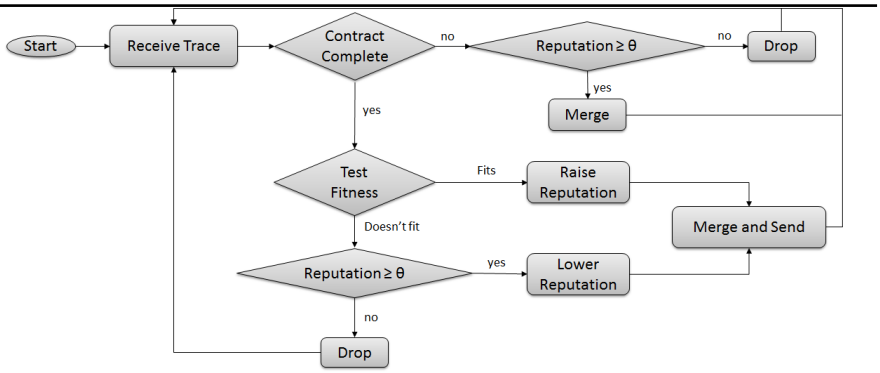


Figure 3.3: Behavior of PICARD Server upon the Reception of a New Execution Trace

is transformed into a clustered execution graph, in which system calls are grouped to form graph nodes and each edge between nodes represents the transition from a node to the next one. From a collection of these graphs, we derive a probabilistic contract that describes quantitatively the expected behavior of the app. In the following, we present step-by-step the generation of a probabilistic contract starting from the acquisition of the execution traces.

3.3.1 ActionNode

To properly represent the contract, we introduce the notion of *ActionNode* [63], which is a cluster of related system calls that (i) are consecutively issued and (ii) are bound by some relation to represent an *action*, i.e., a high-level operation. In general, any relation among system calls can be used, e.g., any partition of all the subsets of system calls. However, in PICARD we only consider those relations that produce a meaningful action. For instance, an ActionNode can be composed by the system calls performed consecutively on the same file, where the relation expresses the fact that all these system calls work on the same file descriptor to produce a relevant action. As we will show in detail, all the system calls forming an ActionNode represent a sub-graph of system calls, while the collection of ActionNodes modeling the app behavior is in turn a graph.

As an example of ActionNode, see Figure 3.4, consider the sequence of system calls: `open(A) - read(A) - read(A) - close(A)`, where A is the filename. This ActionNode, called `Read_File`, represents at high level the action of reading consecutively data from file `A`: this action requires that, firstly, the file has

to be opened, then data is read in a loop and, finally, the file is closed. Some other examples of ActionNodes are depicted in Figure 3.5. Notice that each ActionNode is actually a graph of system calls. The three examples report, ordered left-to-right, the actions of opening, writing, and closing a file (where the same list of operations is performed at least twice), the action of reading a file and then manipulating the underlying device parameters (probably a socket), and finally the action of reading from a previously opened file.

Several advantages stem when using ActionNodes. In fact, by representing the execution graphs through ActionNodes, the app traces can be seen as a graph whose nodes are high-level actions. This representation is more meaningful and compact than a graph whose nodes are just system calls. In fact, generally a program executes several system calls that, taken as standalone in a trace, only give limited information about the app behavior. This happens because only few system call types are issued by apps repeatedly. In fact, by representing the traces through a graph where each node is a different system call, then the program behavior would be represented by a graph with few nodes and a large amount of edges, which form a full mesh. However, from this kind of graph it is not possible to extract a significant contract capable of representing the probabilistic high-level behavior of the app.

The internal nodes that compose an ActionNode, i.e., system calls, are called *SysCallNodes*. In the following, we consider as SysCallNodes system calls that act on files, namely the `open, read, write, close, ioctl` system calls. The pseudocode for the generation of the graph of ActionNodes from the system call traces is reported in [19]. When traversing the trace of system calls generated by a monitored app, the algorithm checks, for each SysCallNode, if the argument is the same as the previous one. Since the monitored system calls act on files, the argument is the filename (or file descriptor). Then, a new ActionNode is created each time a system call is issued with an argument that differs from that of the previous system call (see Figure 3.6), which can be seen as the parameter of the whole ActionNode. Different and subsequent system calls with the same argument are inserted into the same ActionNode (Figure 3.7). If the system call is the same

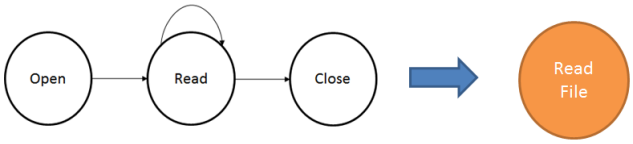


Figure 3.4: Definition of an ActionNode

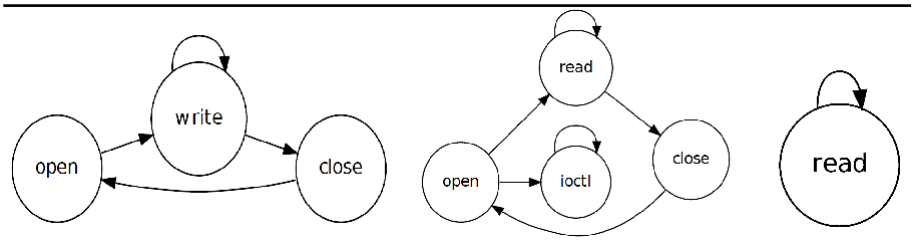


Figure 3.5: Three Examples of ActionNodes

as the previous one, with the same argument, then an edge (between SysCallNodes) is generated that goes back to the same SysCallNode. If the argument is the same, but the system call is different, a new SysCallNode is added to the current ActionNode, provided it has not been already created. In this case, an arc (between SysCallNodes) is added from the current SysCallNode to this existing SysCallNode. Then, each ActionNode is considered as a node representing the high-level operation.

After a new ActionNode is generated, an edge (between ActionNodes) is added from the previous ActionNode to the current one. Self loops may arise because there might be two consecutive actions that are represented by the same ActionNode. Furthermore, a new ActionNode is added only if a similar one does not exist already. To check if an ActionNode already exists, a comparison is made among the internal structure of the new ActionNode and of the existing ones (i.e., the structure of the graphs of the internal SysCallNodes must be the same). Notice that the ActionNodes are oblivious of the filename, meaning that the same cluster of operations performed on two different files generates the same ActionNode.

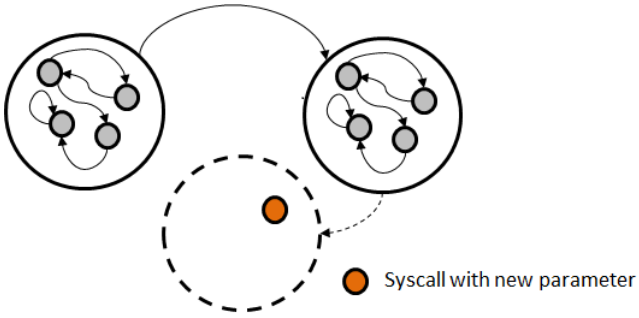


Figure 3.6: Generation of a New ActionNode

```
start, open(10), read(10), read(10), close(10), open(11), read(11), read(11), ↩
    close(11), open(10), read(10), read(10), close(10), ioctl(20), ioctl(20), ↩
    open(10), read(10), read(10), read(10), close(10), open(12), read(12), ↩
    read(12), read(12) , read(12) , close(12) , open(14) , write(14), write↩
    (14), close(14), ioctl(20), ioctl(20), ioctl(20), open(11), write(11), ↩
    write(11), close(11), open(14), write(14), write(14), close(14)
```

Table 3.1: Original Trace of System Calls

In the end, a graph of ActionNodes is generated. In the next step, to define the contract, PICARD takes as input the graph of ActionNodes and outputs a probabilistic automaton. As an illustrating example, Table 3.1 shows a simplified trace of system calls issued by an app where, for the sake of conciseness, only the system call name and the file descriptor are shown.

Figure 3.8(a) reports the graph of system calls representing the trace reported in Table 3.1. The system calls are then clustered on the base of their parameter, in order to extract the ActionNodes presented in Figure 3.8(b). Finally, Figure 3.8(c) depicts the graph of ActionNodes obtained through the algorithm of Figure 3.6 and 3.7.

3.3.2 Traces Analysis and Contract Generation

In the previous section we have seen that by using the notion of ActionNode an execution trace of system calls can be represented as a graph of ActionNodes. In this section we give the formal representation of this type of graph, which is called *labeled multidigraph of ActionNodes* (LMA, for short). We then show how to derive a probabilistic contract from the LMA representing the observed app behavior.

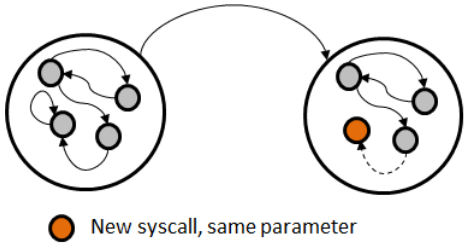
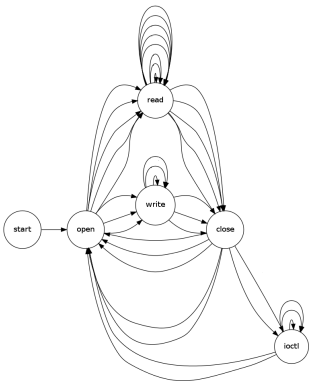
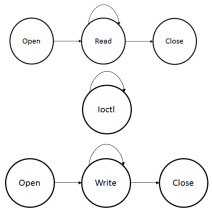


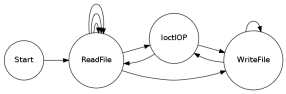
Figure 3.7: Insertion of a New Node in an ActionNode



(a) System call graph



(b) ActionNodes



(c) ActionNode graph

Figure 3.8: Transformation from System Call Graph to ActionNode Graph

Definition 1. *A* labeled multidigraph of ActionNodes *(LMA) is a tuple* (V, I, E, s, t, L)*, where* V *is the finite set of ActionNodes that describe the high-level operations performed by a specific app,* $I \subseteq V$ *is the set of initial vertices from which traces of observations can start,* E *is the finite set of edges,* $s : E \to V$ *is a mapping indicating the source vertex of each edge,* $t : E \to V$ *is a mapping indicating the target vertex of each edge,* $L : E \to \mathtt{T}$ *is a labeling function from edges to the trust domain* T *such that* $L(e)$ *denotes the trust level of the user who observed the execution associated to the edge* e*.*

Notice that there can be both multiple occurrences of edges with the same source and target vertices and edges insisting on the same vertex. While the structure of the LMA is given by the user executing the specific app, the trust related labeling is defined by the central server – when receiving the LMA – on the basis of user's reputation. Then, from the union of the LMAs collected by the central server from all the collaborative users, which in turn is a LMA, a probabilistic contract is built. It is worth noticing that any new multidigraph that is submitted to the central server by a certain user is joined to the current one stored in the central database, and from the union of the two a novel probabilistic automaton is recomputed.

Formally, the probabilistic contract is defined in terms of a structure called *labeled probabilistic automaton* (LPA, for short) which is built from the LMA.

Definition 2. *A* labeled probabilistic automaton *(LPA) is a tuple* (V, I, P, T) *where* V *is the finite set of states,* $I \subseteq V$ *is the set of initial states,* $P : V \times V \to [0, 1]$ *is the transition probability function satisfying* $\forall v \in V : \sum_{v' \in V} P(v, v') = 1$*, and* $T : V \times V \to \mathtt{T}$ *is the transition trust function.*

From the LMA (V, I, E, s, t, L) we derive the corresponding LPA (V, I, P, T), where P and T are defined as follows:

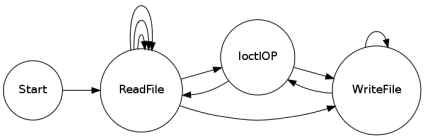
- $P(v, w) = \begin{cases} p \text{ if } \exists e \in E : s(e) = v \wedge t(e) = w \wedge p = \frac{mul(v,w)}{mul(v)} \\ 0 \text{ otherwise} \end{cases}$
  where $mul(v, w)$ is the multiplicity of the edges from v to w in E and $mul(v)$ is the number of outgoing edges from v in E.
- $T(v, w) = f\{L(e) \mid s(e) = v \wedge t(e) = w\}$ where f is any function that, applied to a multiset of trust values, returns a trust value.

On one hand, notice that edges of the LMA with the same source and target vertices collapse into a unique transition in the LPA. Multiplicities of these edges in the LMA are used to compute the transition probability in the LPA. On the other hand, the trust level associated to a transition of the LPA derives from a combination of the trust levels of the edges of the LMA contributing to the transition. For instance, provided that the trust domain is totally ordered and numeric, the combination can be formalized through a mathematical function like, e.g., *min*, *max*, and *avg*. In particular, by using function *max* we assume that a transition is trusted as the most trustworthy user who observed its execution.

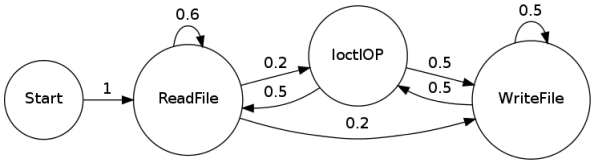
For instance, consider the transformation depicted in Figure 3.9, related to the same example of Figure 3.8. For the sake of simplicity, we abstract away from the information related to trust. Let us concentrate on the `ReadFile` vertex of the LMA, which has five outgoing edges, i.e., three self-loops, one edge directed to

the `IoctlOP` vertex, and one edge directed to the `WriteFile` vertex. The corresponding state in the LPA has three outgoing transitions, i.e., one self-loop with probability $\frac{3}{5}$, expressing that three out of five edges departing from the vertex are self-loops, one transition towards state `IoctlOP` and one transition towards state `WriteFile`, both with probability $\frac{1}{5}$. Notice that the self-loop in state `ReadFile` summarizes three edges that may derive from the multidigraphs of different users. Hence, the trust level associated to such a transition results from a combination of the reputations of these users.

At run-time, the quantitative compliance of the app behavior with the contract is evaluated by comparing the frequency of the observed execution traces with respect to the probability distributions of the expected behaviors extracted from the LPA (V, I, P, T) associated to the contract. These distributions refer to the probability, when starting from any state $v \in I$, of observing distinct finite executions, which we call *longest distinct paths*. More precisely, a longest distinct path starting from v is a finite path that traverses every state in the path at most once, except possibly for the last one, which is either an absorbing state with no outgoing transitions or the unique state of the path visited twice. Intuitively, a longest distinct path describes a maximal finite observation of non-repeated behaviors. The motivation behind the choice of considering the longest distinct paths is that any path including two occurrences of the same state (different from the last one) can be actually viewed as the concatenation of two distinct observations. Hence, any execution trace that is observed at run-time represents a sequence of longest distinct paths,



(a) ActionNode graph



(b) Contract

Figure 3.9: From LMA to LPA

each one associated to its own frequency. As mentioned before and as we will see in the next section, such frequencies have to be compared with the probability distribution associated to the set of longest distinct paths of the LPA, which are calculated formally as follows.

Definition 3. *Given a LPA* (V, I, P, T)*, a* longest distinct path *starting from state* $v_1 \in I$ *is any finite sequence of states* $v_1 \dots v_n$*, with* $n \geq 2$*, such that:*

- $P(v_j, v_{j+1}) > 0$ *with* $1 \leq j < n$*;*
- $\forall v_j, 1 \leq j < n$*, there does not exist* $k \neq j, 1 \leq k < n$*, such that* $v_j = v_k$*;*
- *either* $\exists v_j, 1 \leq j < n$*, such that* $v_j = v_n$*, or* v_n *has no outgoing transitions.*

As usual, the probability of a path $v_1 \dots v_n$ is computed as the product of the probabilities of the transitions forming the path:

$$Prob(v_1 \dots v_n) = \Pi_{i=1}^{n-1} P(v_j, v_{j+1})$$

By following classical results related to probability distributions and measures for probabilistic systems [73], it is natural to construct a probability space over the set of distinct paths starting from any state $v \in I$ (in this set neither of the given paths is a prefix of another in the set). In particular, we have a unique probability distribution over the set of distinct paths of the same length starting from v, whose overall probability sums up to 1. Analogously, the finite set of longest distinct paths starting from v is measurable as well and defines a probability distribution.

As an example, reconsider the LPA of Figure 3.9 and examine the possible observations starting from state `ReadFile`. The related set of all the longest distinct paths is reported in the first column of Table 3.2, which is constructed by exploring every possible sequence of transitions starting from state `ReadFile` and satisfying the three conditions of Def. 3. By calculating the related probabilities as discussed above, we obtain the probability distribution reported in the second column of Table 3.2 (RF stands for `ReadFile`, IO for `IoctlOP`, and WF for `WriteFile`).

Then, by exploiting the trust information labeling the transitions of the LPA, we associate a trust level to each longest distinct path. Intuitively, such a trust level shall be a combination of the trust values associated to the transitions forming the path. Similarly as previously argued for the definition of function T in Def. 2, several functions are candidates to characterize such a combination. However, in order to favor a conservative and cautious approach, we argue that the trust level of a path should not be higher than the trust level of its weakest transition. For instance, we advocate the use of function *min*, i.e., the trust level of a path depends on the least trustworthy transition forming the path.

To summarize, the probabilistic behavior specified by the contract is given by the set of probability distributions associated to the longest distinct paths starting from the potential initial states of the LPA, each of these paths equipped with the related trust level. A fundamental issue is related to determining whether such a set represents a complete, exhaustive contract for the app, i.e., it describes all the possible expected behaviors of the app together with the related probabilities of being observed. This is not a trivial issue that can be solved by comparing the obtained contract with the set of potential behaviors determined, e.g., statically, and that characterize the app functionalities, because the contract includes quantitative, probabilistic information requiring some form of validation. In other words, we need an objective condition establishing that the quantitative information extracted from the LPA is sufficient to characterize the app behavior and, therefore, the obtained contract is ready for distribution to the collaborative users. As anticipated formerly, this level of acquired knowledge shall be stated by a *completeness* index, which is based on the accuracy of the quantitative information characterizing the contract. To this aim, we notice that initially the construction of the probability distributions of the longest distinct paths passes a transient phase during which they change by virtue of the LMAs added by users. Ideally, after such a transient phase, these distributions shall reach a steady state, which would represent the completeness proof. In order to approximate such a theoretical result, we assume that the contract is complete and ready for distribution whenever the last n consecutive trace executions received by the central server contribute to alter the probability distributions up to a tolerance threshold ϵ. The choice of these two parameters defines the tradeoff between accuracy of the obtained contract and time waited prior deployment of the contract. We point out that reducing the delay between app release and contract release is important from user's standpoint, who needs to check the compliance of the own installed app with respect to the con-

Path	Probability
RF-RF	0.6
RF-IO-RF	0.1
RF-IO-WF-IO	0.05
RF-IO-WF-WF	0.05
RF-WF-IO-RF	0.05
RF-WF-IO-WF	0.05
RF-WF-WF	0.1

Table 3.2: Longest Distinct Paths Starting From State `ReadFile` (RF) of Figure 3.9

tract as soon as possible. On the other hand, it is also worth recalling that even a complete contract continues to be adjourned by further contributions provided by users. These observations suggest to reduce the waiting time and augment the level of approximation, by acting on parameters n and ϵ, in order to anticipate as much as possible the contract release by sacrificing its correctness as little as possible.

3.4 Contract Compliance

Once the contract has been built, it can be used to verify the compliance of different versions of the same app on distinct smartphones where these versions have been installed. A non-compliant app is an app exhibiting a behavior different from the one declared in the contract. More specifically, an app is non-compliant when it performs one or more operations not included in the contract, i.e., *functional misbehavior*, or when a sequence of operations is observed with a probability distribution appreciably different from the one associated to the same sequence as defined in the contract, i.e., *non-functional misbehavior*. Functional misbehaviors are captured by solving a subgraph isomorphism problem [87] between the functional projection of the LPA underlying the contract (obtained by removing probabilities from the transitions) and an analogous version of the LPA that is extracted from the observed behavior of the app at run time. On the other hand, non-functional misbehaviors require the analysis of quantitative behaviors. In the following, we consider such a case by showing how to compare quantitatively the probabilistic behaviors of the contract and of the app at run-time.

Let C be the probabilistic contract of an app A. We want to verify the compliance of A', a possibly different version of A, against C. To this end, we monitor the behavior of A' by progressively building the ActionNodes resulting from the observations and then extracting both functional and nonfunctional characteristics of the longest distinct paths that are obtained through the method discussed in the previous section. Then, the resulting probability distributions are compared with those forming the signed contract C. The comparison among probability distributions is typically estimated by means of similarity or distance measures [36]. In particular, we propose the usage of two of these metrics.

The first metric we consider is based on the *Pearson's Chi Squared Test* [102] for estimating the consistency of the behavior of A' with respect to the probability distributions characterizing C. The Pearson's Chi Squared Test belongs to a family of distance measures containing the Squared Euclidean distance $\sum_i (x_i - y_i)^2$, where x_i and y_i denote the probability values of the i-th pair of elements under

comparison, and is used in statistics to verify if a sample is statistically consistent with respect to a known probability distribution. In our setting, the events generated by A' represent the statistical sample, whilst the contract describes the known distribution associated to the observable longest distinct paths.

At run-time, by monitoring the execution of A' we incrementally build the probabilistic behavior of the resulting longest distinct paths. Given a certain state v, to which the contract C associates n possible longest distinct paths, let us denote with O_i the probability associated to the observation of the i-th longest distinct path during the execution of A', and with E_i the expected probability of the same path as stated by the contract C. Then, the chi-squared χ^2 is computed according to the following formula:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

To verify the chi-squared null hypothesis, i.e., the behavior of A' has a distribution consistent with the one described in the contract C, the test statistic is drawn from the chi-squared distribution. If the computed probability is higher than conventional criteria for statistical significance the null hypothesis is not rejected, i.e., the behavior is compliant with the contract and the app should not be considered repackaged.

By following such an approach, however, trust is not taken into account. In order to consider this additional dimension, we propose to discount the result of the i-th comparison by a factor proportional to the trust associated to E_i. Formally, we normalize the trust values to obtain a value in the interval $[0, 1]$ and then we multiply the normalized trust associated to the i-th longest distinct path by the term $\frac{(O_i - E_i)^2}{E_i}$. In this way, we emphasize that differences about untrusted observations are more tolerated with respect to analogous differences about trusted observations.

We point out that, by virtue of the chosen measure, a behavior with non-zero probability in A' that, instead, does not occur in C, cannot involve any comparison, to emphasize that in this case the observed behavior is non-compliant with the expected behavior specified by the contract. Notice that such a misbehavior would be revealed also in a purely functional setting. However, while in the functional setting such a result is simply binary – a new, unexpected behavior occurs that is not allowed by the contract – in the quantitative setting we can estimate the probability of observing such a misbehavior, which, if negligible, could be tolerated to some extent.

To this end, we need a distance measure that allows PICARD to compare distributions over sets of elements some of which could be associated to null probabil-

ities. A candidate metric is the Lorentzian measure, which combines the absolute difference with the natural logarithm:

$$Lor = \sum_{i=1}^{m} ln(1 + |O_i - E_i|)$$

where the term 1 is added to ensure non-negativity and to eschew the log of zero, while m refers to the cardinality of the union of the sets of longest distinct paths observed in A' and those defined in C.

Finally, it is worth noticing that the statistical analysis for contract compliance is employed to verify *a posteriori* the statistical similarity between the probabilistic behavior expected by the contract and the probabilistic behavior observed by every collaborative user who has sent a proper execution trace to the central server in order to generate the contract. Obviously, the same test is executed also for each further execution trace received by the central server even after the generation of the probabilistic contract. The objective of such a similarity analysis is to adjust the reputation of every collaborative user proportionally to the fitness of the execution trace provided by the user with respect to the contract. In particular, strong dissimilarities identify possibly malicious execution traces provided by users who, deliberately or not, generated false or inaccurate multidigraphs. Hence, to keep track of the result of such a similarity analysis, the reputation of the users involved is increased (resp., decreased) if the distance measure is below (resp., beyond) a given trust (resp., untrust) threshold, by an amount proportional to the difference.

3.5 Experimental Results

PICARD has been implemented for Android devices, mainly because of the wide distribution of this mobile OS and its openness. Android is in fact an open source project, based on a custom Linux kernel, which also allows for building customized versions of the OS, called custom ROMs. The execution traces of system calls are captured by using a kernel module, which hijacks the traced system calls and writes on a file the system calls along with the relevant parameters (e.g., the file descriptor). In the current version, the tested smartphones need to be rooted, since command `insmod`, which is used to run the tracing kernel module, can be issued only by a super user in the Android kernel. After hijacking the called system calls, PICARD passes the collected information to the application level through a file buffer shared between the two levels. At the application level, PICARD checks continuously the shared buffer and stores the monitored system call traces.

In order to avoid that an app can interfere, possibly maliciously with other apps running on the mobile device, the Android Linux Kernel enforces isolation cleverly exploiting the multi-user feature. In fact, each app runs in a Dalvik Virtual Machine (DVM), which is an optimized version of the Java Virtual Machine (JVM) acting as a sandbox for the app. Every DVM (i.e., app) receives a Linux User-ID and is treated as a Linux user by the kernel. Hence, each DVM has an `home` directory and its own memory space. Moreover, every app can launch Linux processes and threads whose owner is the associated DVM, identified by the User-ID. The User-ID is extracted from the package of the specific app and we exploit this feature to retrieve the actions performed by a specific app at the kernel level.

To prove the effectiveness of PICARD in discerning between genuine and trojanized apps, we tested our approach on a set of Android apps. The aim of PICARD is to recognize if an app is repackaged, i.e., not compliant with a probabilistic contract. To this end, we have analyzed a set of apps of which we were able to analyze both the genuine and the repackaged version. In the following, we report details about eight representative apps that we consider meaningful. The app contracts have been built running the genuine apps, downloaded from the official Google Play market, on real devices (Samsung Galaxy Nexus with Android 4.0) collecting traces of real usage of different users. We report in Table 3.3 the list

Application	**Traces**	**Length (min)**	**ActionNodes**	**Edges**	**Misbehavior**
`TicTacToe`	100	10	7	30	Send SMS
`LunarLander`	50	10	11	37	Send SMS
`BaseballSuperstar`	100	15	11	46	Geinimi
`AngryBirds`	100	10	21	103	Geinimi
`K-Launcher`	50	15	13	52	KMIN
`Jewels`	50	15	18	78	PJAPPS
`Hamster Super`	50	10	9	35	YZHC
`Tower Defense`	50	10	18	49	Geinimi

Table 3.3: Data About Traces Used to Build the Contracts

of tested apps including the number of collected traces, their length, the amount of action nodes and edges of the app contract and the kind of misbehavior or malware name found in their repackaged version. The first two applications are distributed as sample, together with the Android SDK. Thus, modifying the source code to introduce a misbehavior is simple. The other apps are real repackaged apps found in the wild. In particular, they have been downloaded either from a

repository of malicious applications[1], or from an unofficial app market[2]. In all the tests, PICARD has been effective in recognizing traces coming from all the malicious apps as non-compliant with the contract. For the last seven apps of Table 3.3, functional misbehaviors have been detected through the comparison between the contract resulting from the generated ActionNodes and the behavior resulting from several execution traces of the repackaged versions. Only for the `TicTacToe` app the functional analysis was not enough, while statistical analysis was necessary to reveal the malware.

Path	E_i	O_i
5-3-1-1	0.00005476	0
5-3-1-3	0.0002	0
5-3-1-5	0.000001	0
5-3-2-2	0.00188	0
5-3-2-3	0.0034	0.0003
5-3-2-5	0.00002	0.000026
5-3-3	0.0004	0.12
5-3-4-1	0.0000012	0.0012
5-3-4-3	0.00014	0.0054
5-3-4-4	0.000189	0.0024
5-3-5	0.0031	0.0015
5-3-7-3	0.0000156	0.00075

Table 3.4: Comparison Related to Some Longest Distinct Paths From state 5

In Figure 3.10 we report the contract of the Android app `TicTacToe`, where transition probabilities are omitted for the sake of readability. We emphasize that the set of execution traces used to generate this contract is representative of the app behavior and, therefore, adequate to generate a complete contract. In particular, with respect to the completeness criteria discussed in Section 3.3, we point out that the last of these traces contributes to changes in the probability distributions specifying the contract below the tolerance threshold $\epsilon = 10^{-5}$. It is also worth noticing that traces longer than the ones collected are not representative of the real usage of the app. In fact, it is unlikely that a user uses continuously one of the reported apps for more than 10-15 minutes.

[1] http://contagiominidump.blogspot.com/
[2] appbrain.com.

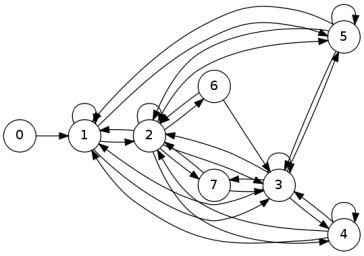


Figure 3.10: Probabilistic Contract of TicTacToe

Afterward, we extracted from a trojanized version of `TicTacToe` 10 traces of a time span of 15 minutes. This trojanized version of TicTacToe sends an SMS message to a phone number each time the user opens the activity to change the graphic style of the game. This action has been triggered at least once for every monitored trace. As an example, we report in Table 3.4 a list of some longest distinct paths starting from state 5 and the related probability distribution both in the contract and in one of the monitored traces.

To verify if the monitored traces are compliant with the app contract, the statistical tests of χ^2 and Lorentzian measure have been used as described in the previous section. The contract of the app includes $m = 7$ probability distributions, as every state in the LPA is a potential initial state, except for node 0 that represents a fictitious initial state. The two statistical tests are performed for all these probability distributions for each monitored trace. The observed behavior of the app is compliant with the contract if the null hypothesis of the test is verified. For the χ^2 test, the null hypothesis h_0 that has to be checked for each probability distribution $i \in \{1, \ldots, m\}$ is $\chi_i^2 \leq \gamma$, where γ is the critical value of the χ^2 distribution for one degree of freedom (as we are considering one monitored trace) and a significance level α representing the desired tolerance (which is a configurable parameter of the framework). As an example, by computing the χ^2 test on the distribution of longest distinct paths outgoing from state 5, which represents the action node `read(A)-read(A)`, we obtain a non-compliance result. For instance, the value returned by the test for a representative monitored trace is $\chi_5^2 = 37.22$. Notice that the critical value corresponding to tolerance $\alpha = 0.999$ (which is a very high tolerance) is $\gamma = 10.828$, which is much lower than the value returned by the test. Thus, the null hypothesis h_0 is not verified and the monitored trace is correctly considered non-compliant with respect to the contract.

By performing the Lorentzian test on the same data, the obtained result is $Lor_5 = 0.13$. In order to favor a correct interpretation, we compare this value with

the ones obtained from monitored traces belonging to the genuine version. The highest value returned from a genuine trace is $Lor = 0.0012$, which is two orders of magnitude lower than Lor_5.

As far as performance aspects are concerned, we report the overhead of the testing of PICARD executed on a Samsung Galaxy Nexus with Android 4.0. The measured overhead of the PICARD app is 7% of the CPU usage and 3% of RAM occupation when performing compliance check with the contract. The PICARD app causes less overhead when simply collecting traces and sending them to the PICARD server. The overhead of the kernel module for system call monitoring is 5% for CPU usage while RAM usage is negligible.

3.6 Related Work

In the literature, system call analysis has been proposed in several systems to monitor and detect malicious behaviors. One of the first network intrusion detection systems (IDS) based on state transition graphs analysis is NetSTAT [124]. Analysis of system calls with Markov models have formerly been performed on other operating systems. For instance, Hoang and Hu [79] propose a scheme for intrusion detection. This model is based on system calls and hidden Markov models and is able to detect efficiently denial of service attacks. Maggi et al. [93] present another model relying on system calls and Markov models to detect intrusions. Their system analyzes the arguments of the system calls but is oblivious of the system call sequence. System call sequences and deterministic automata have been used by Koresow [85] to detect anomalies, which are revealed when system call sequences differ from an execution trace known to be secure. *Crowdroid* [34] is an Android-based IDS that is based on the number of system calls issued by an application. Misbehaviors are identified by applying computational intelligence techniques.

A behavioral analysis of Android applications at the system call level is presented in [105]. The authors propose a framework called CopperDroid that discerns good behaviors from bad ones. Copperdroid tries to automatically stimulate malicious applications to misbehave through instrumentation. The analysis of behaviors is automatic, which means that the behavior of the application stimulated by user interaction is not considered. Another approach that aims at classifying malicious behaviors is presented in [95], which aims at finding similar sub-graphs describing common behaviors of malicious apps. A classifier is then used to determine whether an app is malicious or not. The approach requires each app to

be run in a sandbox, which may alter the execution of some actions that are usually performed on real devices. A framework for analysis of Android apps is presented in [132]. The proposed framework emulates in a virtual sandbox all the components of the Android framework and can be used to analyze the behavior of Android apps by collecting traces at the Dalvik level. This framework can be complementary to the PICARD approach and as future work it would be worth considering the possibility of using the Dalvik level traces for higher level ActionNodes. The framework proposed in [81] analyzes the behavior of Android applications by running them on emulators in desktop environments. Each application is forced to execute with as many input as possible, in order to discover as many hidden behaviors as possible. The security analysis is then performed offline on the functions called at the *smali* (bytecode) level. This approach is different from the one of PICARD, which aims at finding discrepancies between the application behavior and the contract. Thus, the concept of "misbehavior" in PICARD is more general, which should be more effective in detecting zero-day attacks.

Some Android security frameworks try to protect the system by monitoring the communication level and defining security policies. One of these systems is CRePE [41], which allows the definition of context based security policies. Another typical approach to monitor Android app behavior is called tainting. In this approach sensitive data flow is tracked to check the apps that are able to access a specific piece of information. Examples of this approach on Android are [23] and [68]. The tainting approach is more aimed at detecting privacy leakage, whilst the PICARD approach addresses a more general concept of misbehavior. In [33] a framework based on SELinux is proposed to enforce security policies on Android devices and to tackle common misbehaviors performed by malware. The approach is more focused on the application of security policies than on detection of malicious apps. Moreover, a custom version of the operating system is required. *Aurasium* [129] is another security framework for Android devices, which is able to enforce security policies through repackaging of all the installed apps. Aurasium forces the apps running on the device to call modified APIs, which are used to perform security checks. Aurasium is aimed at enforcing security policies instead of the detection of repackaged apps.

Another approach that exploits application contracts to monitor the behavior of mobile applications is given by the Security-by-Contract framework [64]. This framework has been extended [42] and applied in several ways and in different scenarios [65, 76]. Differently from PICARD, the security by contract framework does not perform intrusion and malware detection, but matches the application behavior with a security policy. Thus, malicious behavior that is not specified in

the security policy is not considered. Referring to quantitative models, probabilistic contracts have been firstly introduced by Delahaye et al. [50, 52, 51] for analyzing reliability and availability aspects of systems. Their approach to contract definition and compliance check is static and is not related to practical applications in the field of intrusion detection. On the other hand, we have shown that PICARD is based on a dynamic approach using statistical analysis that turns out to be effective in detecting trojanized apps. Bielova and Massacci [28] present a notion of distance among traces characterizing enforcement strategies by the distance from the original trace. This approach is generic and, differently from PICARD, it does not consider low level actions, nor proposes real applications. The chi-squared test has been used to detect anomalies in several different fields. For instance, an application to network traffic analysis is presented by Ye and Chen [134].

The work presented in [15] describes a system to classify malicious apps on the base of the API calls they perform. The approach uses a classifier that statically analyzes features related to the called methods, with a particular attention to intercommunication constructs. This approach is effective in foreseeing malicious misbehaviors which are functional, albeit it is less likely to discover non-functional misbehaviors.

3.7 Conclusion and Future Work

In this paper we have proposed PICARD, a collaborative framework for generating and checking Android apps' probabilistic contracts. PICARD performs analysis of the app behavior at run-time, by building the contract dynamically. To this aim, we have introduced the concept of ActionNode in order to describe the behavior of apps through clustered graphs and probabilistic automata. PICARD discerns between genuine and trojanized apps, by revealing both functional and non-functional misbehaviors.

The dynamic approach used by PICARD is more specific with respect to an approach relying on the integrity check based on the checksum of the `apk`s. In fact, if an app is updated, e.g., by changing only some of its data, such as an utility app that changes a background picture, the app still performs the same actions as the former version and, hence, has the same behavioral contract but, however, the checksum is different. Moreover, in PICARD the analysis is conducted through statistical tests performed on the app execution traces described in terms of probabilistic automata. Hence, the PICARD approach does not differentiate the behaviors in "known" and "unknown" only, but it is also able to distinguish between likely and unlikely behaviors. The dynamic analysis of PICARD does not require

the program to be decompiled. Moreover, since the contract is built by monitoring real user behaviors, it is possible to detect misbehaviors that may not be noticed through static analysis. Thus, PICARD can be viewed as a framework complementary to static analysis systems.

The effectiveness of PICARD depends strongly on the cooperation of several users, thus motivating the use of a reputation system stimulating collaborative, honest behaviors. In particular, the proposed approach is based on a computational notion of trust inspired by the Jøsang model [82], which keeps track of the history of the user behavior, by increasing reputation for each correct report and by decreasing it otherwise. A similar approach is presented in [27]. Moreover, in our approach the quantitative behavior of the app reported by users is weighted by the reputation of the user, while the notion of contract compliance is based on the comparison between probability distributions through a distance metric. Employing reputation-based weights and similarity tests to rate the credibility of the reported feedback is not a completely naive approach and is used, e.g., in the TrustGuard framework [117], which defines heuristics to mitigate dishonest feedback. In order to discourage false or inaccurate feedback, more sophisticated cooperation incentives can be used (see, e.g., [72, 133]), possibly based on some form of remuneration (see, e.g., [29]).

The validation of the trust configuration policies and parameters discussed in Sections 3.3 and 3.4 is left to sensitivity analysis in future work. We also plan to extend the experimental studies by verifying whether the reputation system of PICARD is effective to isolate selfish behaviors due to non-collaborative users and to detect and punish (possibly orchestrated) malicious behaviors due to collaborative users who deliberately cheat by providing false feedback during the contract generation.

Further extensions are related to the notion of ActionNodes, which, in the current implementation, are built on system call graphs. In particular, the concept of ActionNode can be extended including complex nodes that describe higher level actions, such as “Send text message”. Finally, we also plan to extend the experiments on a larger set of apps, with the possibility of including an automatic analysis approach, as in [97] and [91].

4

Introducing probabilities in contract-based approaches for mobile application security

4.1 Overview

New generation mobile devices (*e.g.*, smartphones and tablets) are becoming day-by-day more powerful and popular. The growth in computing power, ubiquitousness and capabilities of these devices has been parallelized by the growth of available applications, specifically developed for smartphones and tablets. However, these applications may be not completely secure. In fact, malicious developers strive to design and deliver applications that may damage both users and devices. In particular some applications may hide a Trojan horse that, even if it looks unharmful, in background it performs malicious actions that the users did not expect to happen.

The current security model, which rules (i) if an application can be safely installed on the device, (ii) what kind of actions the application may execute once installed, still suffers from several weaknesses, in particular in its capacity of expressing proper *contracts*. Semantics of current security models is too naïve since it is either based upon trust relationships or upon statements of purpose. In the first case, users accept to run an application if they trust the provider. In the second one, providers state the security relevant actions performed by an application and it is up to the users to decide whether run the application if they consider these operations safe. In the former case the trust level of the trusted entity also determines the code privileges, essentially relegating an application into the "all or nothing" policy, while in the latter case the semantics is too-coarse grained (*e.g.*, Android permissions) or hardly usable. For example, in the Android system, security relevant actions are declared through permissions, which are difficult to understand for average users.

In this paper, we introduce probability aspects into the workflow of two contract-based approaches developed for mobile devices, namely the Security-by Contract [64] (S×C) and the Security-by-Contract-with-Trust [43, 42] (S×C×T) frameworks. These two approaches integrate several security techniques to build a chain of trust, which, in the end, ensures that the downloaded application will execute only security actions that are allowed by the user's policy. To this end, we introduce a probabilistic description of the behavior of an application and a more expressive version of the user's security requirements. Indeed, the current models only permit the definition of a set of allowed actions, *e.g.*, the Android permission system (first box of Figure 4.1). More expressive policies which take in account a possible action hystory are modelled through automata that represent allowed executions.

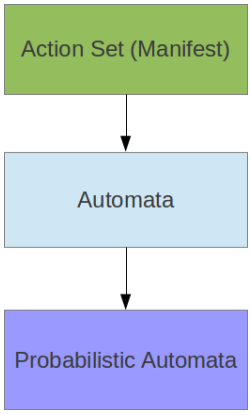


Figure 4.1: Graphical representation of the improvement in policies expressiveness.

We propose a probabilistic automata-based model that enables the developers to define more expressive contracts through probabilistic clauses, *e.g.*, how often a security-relevant action may happen. The same expressiveness is given to users to specify security policies. Since we include probabilistic clauses in the specification of contracts and policies, the security mechanisms involved into the workflows of S×C and S×C×T has to be redefined. Hence, we present a new workflow for both S×C and S×C×T framework in which each module is updated to support probabilistic functions. The advantage of using probabilities is the possibility of describing more realistic usage scenarios for an application. In fact, many appli-

cations depend on user inputs or context information and it is difficult to define realistic policies based upon boolean conditions only. In these models, all the possible execution paths are considered legal. Hence, a low-probability operation is considered valid even if performed several times. For this reason, we introduce probabilities in the definition of security clauses to define more fine-grained contracts and policies. These descriptions better fit real application use cases and can be defined without alteration of the Security-By-Contract-with-Trust workflow. Finally, we propose an extension to the Android permission system, which includes the security properties of Probabilistic Security-By-Contract with Trust in the most popular operative system for mobile devices.

The paper is structured as follows. Section 4.2 introduces the main concepts of contract-based approaches and briefly recalls the Security-by-Contract and the Security-by-Contract-with-Trust frameworks. In Section 4.3, we propose a probabilistic version of both Security-by-Contract and Security-by-Contract-with-Trust. Section 4.4 presents an application of the proposed approach in Android systems, proposing some extension to the current framework. In Section 4.5, we discuss some related work, while Section 4.6 briefly concludes.

4.2 Contract-based Approaches

Contract-based approaches have been developed for mobile devices, such as the Security-by-Contract [64] (S$\times$C) and the Security-by-Contract-with-Trust [43, 42] (S$\times$C$\times$T) frameworks. They integrate several security techniques to build a chain of trust by sequentially applying them to safely execute applications. The three cornerstones of these security frameworks are application *code* A, application *contract* C, and client *policy* P, where a *contract* is a formal, complete, and correct specification of an application security relevant behavior, *e.g.*, security critical virtual machine API call, or critical system calls [77]. A *policy* is a formal complete specification of the acceptable security-relevant behavior allowed to applications executed on the platform [77]. We assume that both contract and policy are syntactically described by exploiting the same language.

The basic idea of a contract-based approach is the usage of the contract for guaranteeing that security aspects are satisfied. More in detail, using the contract, it is possible to check at deploy time, *i.e.*, before the application execution, if the application satisfies the user policy or not. Let $\preceq$ denote the compliance between two of the previous elements. A contract-based approach guarantees that

$$A \preceq C \preceq P \Rightarrow A \preceq P \tag{4.1}$$

In the following, we describe the Security-by-Contract (S×C) and the Security-by-Contract-with-Trust (S×C×T) frameworks as approaches that integrate the described techniques to guarantee security at application execution time.

4.2.1 Towards Security Techniques

Several techniques have been proposed for tackling specific security aspects. Almost all the following are integrated into the S×C or into the S×C×T frameworks.

Application-Contract Matching. It enables statically verification of an application code by using a third-party provided proof and also its validation. The proof is linked to the application code. Verifying the proof validity is more efficient than generating it. The verification procedure follows the steps of the proof and, if all of them are correct, validates its conclusion. Examples of this approach are the *proof-carrying code* [99] and the *model-carrying code* [114] methods.

Contract Policy Matching. It statically analyzes the compliance of a specification, *e.g.*, a contract, with a specified security policy.

Enforcement/Monitoring. The run-time enforcement approach consists of running an application code inside the scope of a *controller* that checks, step-by-step, the executed operations. At each operation, the behavior of the considered application is compared with the consumer policy (*policy enforcement*), and prevents violations by modifying the application behavior at run time, e.g. forbidding non-allowed operations.

This approach differs from monitoring that just observes the behavior of the application and at the end of the executions it could also provide information (*e.g.*, audit, logs) for understanding its behavior, *e.g.*, if the code does not work as described by its contract (*contract* monitoring).

Metrics Manager. Security metrics aim at assessing security threats. For instance, metrics can describe a system in terms of its reputation in a community, number of past, successful interactions or average number of failures per year. Then, these values are exploited for taking security aware decisions. Among the others, *trust*, *risk*, and *probability* aspects are receiving major interest.

4.2.2 Security-by-Contract and Security-by-Contract-with-Trust in a Nutshell

The Security-by-Contract paradigm provides a full characterization of the contract-based interaction. It combines different functionalities in an integrated way (see Figure 4.2). In particular, it includes a module for automatically checking the formal

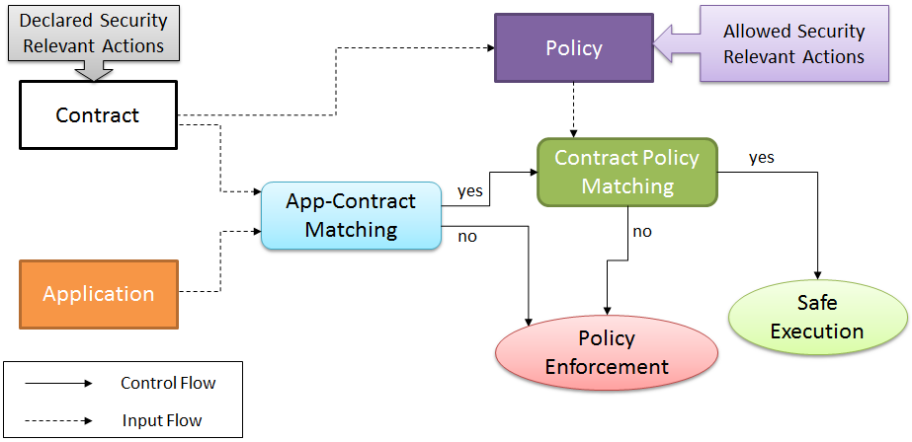


Figure 4.2: The Security-by-Contract process.

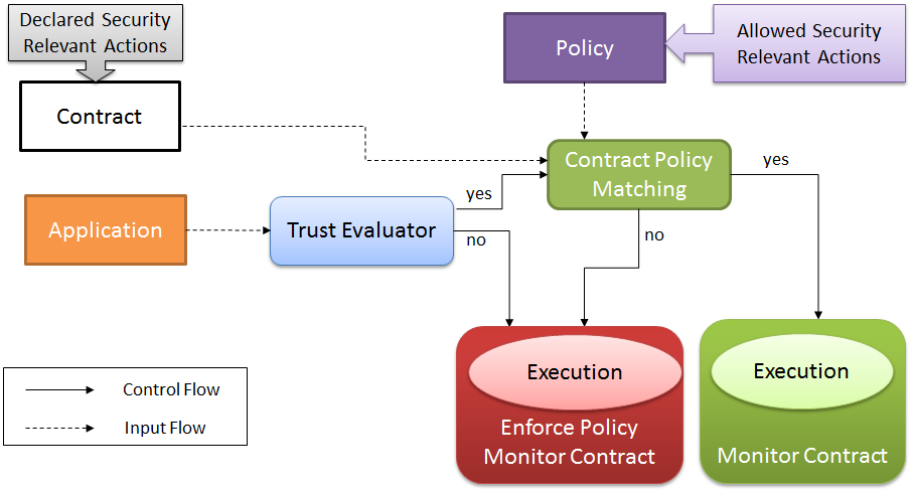


Figure 4.3: The Security-by-Contract-with-Trust process.

correspondence between code and contract (*Application-Contract matching*). If the result is negative, then the monitor is run to enforce the policy (*Policy Enforcement*), otherwise a matching between the contract and the policy (*Contract-Policy Matching*) is performed to establish if the contract is compliant with the policy. In this case, the code is executed without overhead (*Safe Execution*), otherwise the policy is enforced again (*Policy Enforcement*).

Along this research line, in [43, 42] S×C has been extended in order to deal also with the concept of *trust*. The new framework is named Security-by-Contract-with-Trust (S×C×T) (Figure 4.3). S×C×T consists of integrating the S×C paradigm with a monitoring infrastructure for trust management. As a matter of fact, a crucial point of the S×C architecture is the verification of the relation that exists between the application and its contract. Usually, nowadays a mobile application is installed only if its origin is trusted. This means that users can reject or accept the signature of the application provider based upon the trust level. S×C×T extends S×C in two different phases: at deploy-time, replacing the app-contract matching with a *Trust Evaluator* module. This component sets the monitoring state, and at run-time it applies the contract monitoring procedure for tuning the provider trust level. In fact, the S×C architecture has been extended by adding a component for the contract monitoring to check if the contract adheres to the actual execution of the application and, according to the answer, it updates the provider level of trust.

The advantages of these contract-based frameworks are that they are able to identify unsafe applications before and without running them. In particular, using the contract-policy matching functionality, it checks at deploy-time if the declared behavior of the application is compliant with the required policy. This check, along with the assurance that the application code is compliant with the application contract, which is obtained through the application-contract matching module (S×C) or by the trust evaluator (S×C×T), guarantees that the application satisfies the user requirements. Anytime the contract-policy matching finds that a contract is not compliant with the policy, the application is run in a controlled way through the enforcement module. It is worth noticing that the cost, in terms of energy, of running a contract policy matching is much lower than performing the enforcement. Hence, unsafe applications are not run at all by the user and possible unsafe application are run in a controlled way. This leads to an attack risk reduction.

4.3 Probabilistic Security-by-Contract and Probabilistic Security-by-Contract-with-Trust

In this section, we describe a probabilistic version of both S×C and S×C×T architectures. It is worth noticing that, in both cases the original workflow is not changed. Only the components are modified in such a way that, on one hand, they are able to cope with probability metrics and, on the other hand, Equation 4.1 still holds for an appropriate choice of the notion of compliance.

Let us assume that both probabilistic contract and probabilistic policy are expressed through the same formalism.

Probabilistic contract and policy will be modelled as *(substochastic) generative probabilistic automata* [78, 26].

Definition 4. *A* fully probabilistic *or* generative *automata is a tuple* (S, Act, P) *consisting of a finite set* S *of states, a set of actions* Act*, and a* transition probability function

$$P : S \times Act \times S \to [0, 1]$$

A generative automata is said to be stochastic *if*

$$\sum_{a \in Act} \sum_{t \in S} P(s, a, t) = 1$$

for all $s \in S$ *for all* $a \in Act$*. On the other hand, a generative automata is said to be* semistochastic *or* substochastic *if*

$$\sum_{a \in Act} \sum_{t \in S} P(s, a, t) < 1$$

for all $s \in S$ *for all* $a \in Act$*. For* $C \subseteq S$*, we put* $P(s, a, C) = \sum_{t \in C} P(s, a, t)$*. A state* $s \in S$ *is said to be* terminal *iff* $\sum_{a,t} P(s, a, t) = 0$.

Hereafter, we consider generative automata such that for each action there is only one possible transition for each action $a \in Act$.

4.3.1 Probabilistic Security-by-Contract Workflow

Being the Security-By-Contract framework modular, introducing probability metrics implies the substitution of some components with their probabilistic counterpart. The Probabilistic Security-by-Contract workflow is depicted in Figure 4.4.

Probabilistic application contract matching is verified using some static validation techniques able to deal with probabilistic description of behavior. For instance, as *proof carrying code* [99] is used in S×C, here we can use the *Probabilistic Proof Carrying Code*, *e.g.*, [115, 121]. In particular, this method guarantees that, for all possible k-length execution traces whose probability is calculated as $P_k = \prod_{i=1}^{k} P(s_i, a_i, t_i)$, the application is considered compliant if $P_k > \theta_k$, where θ_k is a given threshold value $0 < \theta_k < 1$ dependent from the length of the execution trace.

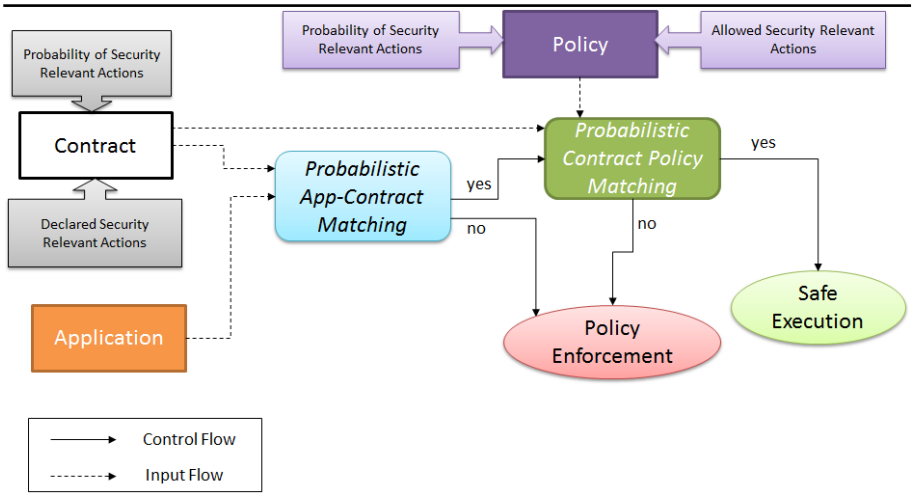


Figure 4.4: Workflow for Probabilistic Security-by-Contract

Probabilistic contract policy matching is performed by checking the compliance between a contract and a policy. According to the level of required accuracy, several relations can be considered in order to verify the compliance between probabilistic contract and policy. In S×C, the contract-policy matching function checks if the contract and the policy are similar. This means that for each action described in the contract, we check if there exists the same action described in the policy and the description of the transition are similar again. Hence, we assume that the policy specifies a rule for each security relevant action, which we call *SecAction*.

Referring to the notion of ε-simulation given in [53], hereafter, we define a slightly different ε-simulation.

Definition 5. *A relation $R \subseteq S \times S$ is a relation of* positive ε-simulation*, where $\varepsilon \in [0,1]$ if whenever $(s,s') \in R$, then $\forall a \in SecAction$, $\forall W \in S$*

$$\sum_{t \in W} P(s,a,t) \leq \sum_{t' \in R(W)} P(s',a,t') \leq \sum_{t \in W} (P(s,a,t) + \varepsilon)$$

where $R(W)$ is the set of all states that are in relation with states in W trough R. We say that s is ε-simulated by s', written $s \prec_\varepsilon s'$, if $(s,s') \in R$ for some relation of ε-simulation R on S.

The idea is that, while the ε-simulation allows a deviation of a values $\varepsilon \in [-1,1]$, here, we are only interested in positive values of ε. Hence, the proba-

bilistic distribution of the contract have to be less that the probability distribution of the policy of, at most, a value ε.

It is worth noticing that, according to our assumptions, having a positive ε-simulation R means that whether $(s, s') \in R$ then, for each action $a \in SecAction$,

$$P(s, a, t) \leq P(s', a, t') \leq P(s, a, t) + \varepsilon$$

and $(t, t') \in R$.

Enforcement of Probabilistic Policies is performed when either the application is not compliant with the contract or the contract is not compliant with the policy. At each step, the enforcement computes the probability that the application performs a specific security relevant action a, starting from the current state s, $P^p(s, a, t)$, where t is the destination state of the transition and p is the expected one stated by the policy. This computation exploits history-based concerning the current execution of the application. The computation of the probability of the execution trace is similar to the one described in the application-contract matching module $P^p_k = \prod_{i=1}^{k} P^p(s_i, a_i, t_i)$. The application is considered compliant if $P^p_k > \theta_k$, where θ_k is the same considered in the application-contract module. The enforcement denies the non compliant operation sequence, ensuring that the policy is correctly enforced.

It is worth noticing that Equation 4.1 holds. Indeed, the fact that $C \preceq_\varepsilon P$ means that $P_k \leq P^p_k$ because

$$P_k = \prod_{i=1}^{k} P(s_i, a_i, t_i) \leq \prod_{i=1}^{k} P^p(s_i, a_i, t_i) = P^p_k$$

Hence, $\theta_k < P_k \leq P^p_k$ Let $\preceq_\Theta$ be the compliance relation used in both application-contract matching and enforcement mechanisms, where Θ denotes the set of threshold values θ_k for any k-length execution trace, and let us consider to use the positive ε-simulation for the contract-policy matching then the following holds

$$A \preceq_\Theta C \preceq_\varepsilon P \Rightarrow A \preceq_\Theta P$$

4.3.2 Probabilistic Security-by-Contract-with-Trust

Let us introduce probability also into the S×C×T architecture. Referring to [43, 42], we consider a trusted marketplace that provides trusted information about the compliance between the application and its contract. The Probabilistic Security-by-Contract-with-Trust workflow is depicted in Figure 4.5.

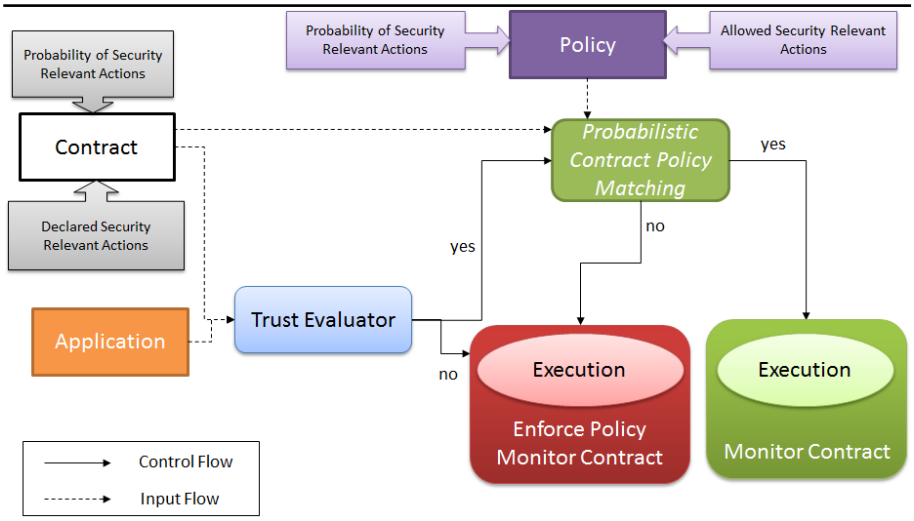


Figure 4.5: Workflow for Probabilistic Security-by-Contract-with-Trust

Referring to the probabilistic S×C workflow, the main difference here is that we do not have the application-contract matching functionality. replaced by the Trust evaluator.

Trust Evaluator assesses the trust level of the application to be compliance with its contract. Also in this case, the compliance relation we consider is the same that we consider in S×C, *i.e.*, $\preceq_\Theta$. Note that even when the developer does not provide a contract for the application. In fact, according to [18], given an application, it is possible to automatically generate its *probabilistic contract*. Let us assume that the marketplace is able to generate it. In this case the level of trust we consider is the level of reliability of the probabilistic contract as complete description of the application behavior.

Probabilistic Contract Generation. The contract is generated by analyzing either application executions or the application code [18], *i.e.*, the application control-flow is analyzed to explore all possible executions, and associating to each execution a probability. The union of all the possible executions constitutes all the possible sequences of states that an application can follow. From these sequences of states, if we only focus on the security relevant actions executed by an application, *i.e.*, SecActions, then the contract is represented by a probabilistic automaton $Q = (V, SecAction, P)$, where the nodes $V = \{v_1, v_2, ..., v_n\}$ is related to the set of states, $SecAction$ is the set of Se-

cActions performed by a specific application, and P is the probabilistic transition function $P : V \times SecAct \times V \rightarrow [0, 1]$ defined as follows: let $mul(v_i, a, v_j)$ be number of times that the action a is executed in the state v_i for reaching the state v_j and let $mul(v_i)$ be the number of output arcs from v_i. Hence, for all state $v_i \in V$,

$$P(v_i, a, v_j) = \frac{mul(v_i, a, v_j)}{mul(v_i)}$$

for each v_j reached by v_i through an action a. $P(v_k, a, v_l) = 0$ is associated to any missing edge between v_k and v_l. It is worth noticing that this model is a generative one.

Probabilistic Contract-Policy matching. Also in this case, matching is performed by checking if there exists a probabilistic simulation relation or an ε-simulation between the probabilistic contract and the probabilistic policy. It is worth noticing that, in any case, a probabilistic automata is returned. Hence, we can exploit the same functionalities we have described in the Probabilistic Security-by-Contract workflow.

Probabilistic Contract monitor. The contract monitor is performed when both the trust level is greater than a policy-defined threshold and the contract policy matching returns a positive answer. This is because, also for Probabilistic Security-by-Contract-with-Trust holds the Equation 4.1. If no violation is detected, then the application worked as expected. Otherwise, we are dealing with a non-compliant contract. In this case, if the contract is provided by the developer, the marketplace has to downgrade the level of trust of the application. On the other hand, if we are dealing with the automatically generated contract, the marketplace has to update the contract by taking into account also the trace that contains the violation.

Probabilistic contract monitor and policy enforcement. Similarly to a pure enforcement framework in Probabilistic S$\times$C, our system guarantees that executions are policy-compliant. However, monitoring contracts during these executions can also provide a useful feedback. Hence, in this scenario, both policy enforcement and contract monitoring are active. To reduce the overhead of the monitoring, the contract monitor is only activated on a statistical base depending on the level of trust of the application.

4.4 Use Case: Android System and Applications

A possible use-case for the probabilistic contract model is represented by the mobile operative system Android.

Android is an *app-based* mobile operative system: it allows users to download and install applications specifically designed for Android devices. The Android operative system is a complex framework that relies on a generic Linux kernel and several libraries written in high level languages that enable the interaction with all the device components. Applications offer several functionalities exploiting the various interfaces and components of the device, by also accessing resources that are security-critical, such as network interfaces, call dialer, SMS manager, or even private data like contact lists, social network passwords, device IMEI and SIM number. Due to the high number of security-critical resources, smartphones and tablets are susceptible to a higher number of security issues than personal computers. In fact, starting from 2009, the number of attacks targeted to mobile devices has strongly increased [83]. In particular, in 2011 and 2012 several malicious applications (malware) have been developed specifically for Android devices.

Android already includes a contract-like system, based upon the concepts of *permissions* and *manifest*. In Android each application comes shipped with a document called `AndroidManifest.xml` (manifest for short) that describes the application components and declares the security actions performed by the application. If an application has to perform some critical operations, such as to access a device resource, or to read/write sensitive information, this has to be declared it in the manifest file. To enforce this contract, a component on the system-side called *Permission Checker* constantly enforces the policy by denying each operation for which the permission has not been declared in the manifest file. Several criticisms have been raised against this system, which results too coarse-grained [131] and too much reliant on user knowledge and expertise [71]. The main problem of this approach is that the acceptance policy for an application's requested permission is "all or nothing", that is, the user cannot accept only a subset of the required permissions.

We argue that is possible to enhance the Android permission system increasing its expressiveness, including in Android the probabilistic Security-By-Contract model. This is discussed in the next subsection.

4.4.1 Extended Manifest and Trust Evaluator

The first step to extend the Android permission system is the extension of the manifest file to include a description of the probabilistic automata. The manifest file is written in eXtensible Markup Language (XML), in which the inclusion of additional data is straightforward. Exploiting this feature of the XML language, we extend the manifest introducing a new xml tag: `<contract_clause>`. This tag contains

a description of the contract probabilistic automaton in Graph Markup Language (GML). GML gives an XML-like description of a graph or automaton and can be easily embedded in an XML document. In this way, the `<contract_clause>` tag is not analyzed by the Android system, which only checks the ordinary permissions.

Hence, in our proposed framework, the manifest file comes as a contract divided in two parts. The first one is filled by the developer and specifies the application components and permissions. The second part, which can be filled either by the developer or directly by a trusted third party, e.g. the *Google Play* market, contains the probabilistic contract. Since in Android the main vector for application distribution is the on-line marketplace, Google Play, the trust relationship is not directly established between users and developers. Developers build a trust relationship with the market, which decides how much it trust the applications coming from a developer. This trust value is added to the manifest file. The trust value is analyzed by the Trust Evaluator component, which decides whether the application is trusted or not.

Example: "The probability that an SMS is sent to a number not in the contact list is lesser than 3%". In Android there are several SMS manager applications which can be downloaded and installed, which may automatically send SMS messages (memo or post-poned sending). These applications should send SMS to known numbers and to unknown ones in a limited amount of times only. This avoids, for example the unwilling subscription to premium services. Simply using the Android permissions `SEND_SMS` and `READ_CONTACTS`, it is not possible to implement such a policy, that even if simple requires a greater expressivity than the one provided by Android. Using probabilistic automaton, the definition of such a policy is straightforward.

4.4.2 Policy Manager, Matching and Enforcement

The Policy Manager is used to specify the security policies, which can be global or per-application. This component presents a simple user interface that allows users to define policies. Moreover this component is able to learn user behaviors concerning security relevant actions, e.g. learning the average of SMS messages sent each day, and to instantiate a proper policy accordingly. In this scenario, the policy manager either receives as input (i) the user-policy, either written in a policy-specification language or even in natural language or (ii) is learnt by monitoring the user behavior. Afterwards, the policy is translated in a probabilistic automata, which can be used for contract-policy matching or policy enforcement.

The contract-policy matching verifies if the manifest extension matches the policy provided by the Policy Manager. This control is executed at deploy time.

If the contract does not match the policy, the user is prompted to decide if she wants to abort the installation, or install however the application with a run-time policy enforcement. The Security-By-Contract-with-Trust enforcement extends the one performed by the permission checker, ensuring that the defined policies are always enforced. Figure 4.6 depicts a workflow of the Security-By-Contract-with-Trust extension on Android devices.

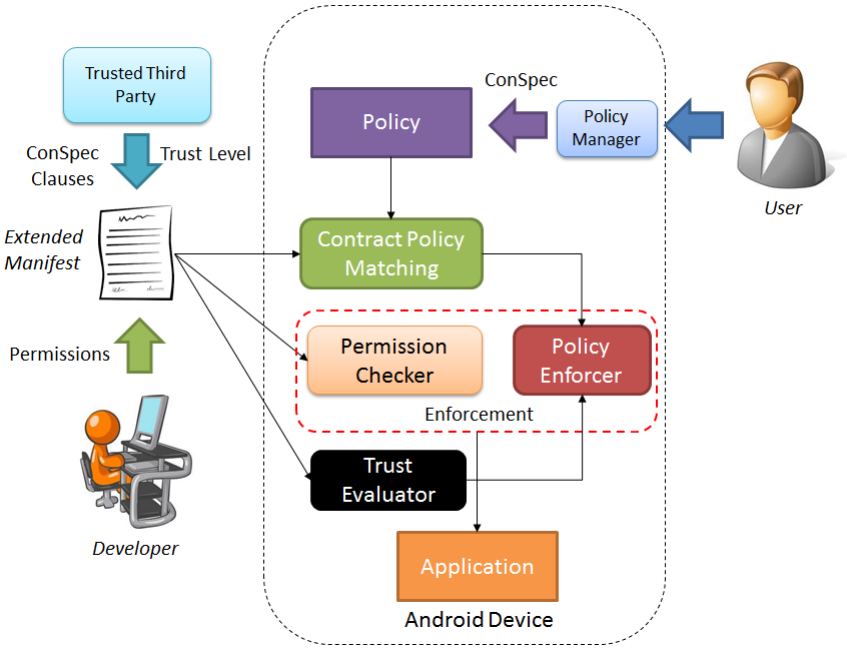


Figure 4.6: Inclusion of Security-By-Contract on Android

Notice that for sake of clarity, the Application-Contract Matching component has not been included in Figure 4.6. This task is demanded to the Trusted Third Party.

4.5 Related Work

In the last decade, the Security-by-Contract framework [64] has been extended and applied in several ways and in different scenarios. For instance, in [43, 42] the extension of the Security-by-Contract with Trust manager has been presented.

Furthermore, in [42] it has been instantiated in a marketplace scenario and in particular, it is integrated with a trust manager able to manage feedback obtained by the monitoring module. Another application scenario is the one of web service [65, 76, 20].

From the quantitative perspective, the problem of finding an optimal control strategy is considered by Easwaran et al. in [66] in the context of software monitoring, where the system is represented as a Directed Acyclic Graph, and where rewards and penalties with correcting actions are taken into account, thus using dynamic programming to find the optimal solution. Similarly, an encoding of access control mechanisms using Probabilistic Decision Process is proposed in [94], where the optimal policy can be derived by solving the corresponding optimization problem. From a different perspective, Bielova and Massacci propose in [28] a notion of distance among traces, thus expressing that if a trace is not secure, it should be edited to a secure trace close to the non-secure one, thus characterizing enforcement strategies by the distance from the original trace they create. A system that exploits system calls to detect non-compliant application is presented in [62]. Referring to probabilistic models, probabilistic contracts has been firstly introduced in [50] for analyzing reliability and availability aspects of systems. The generation of probabilistic contract has been made by analyzing the occurrences of system calls. In [79] a scheme for intrusion detection using probabilistic automata is proposed. This system exploits system calls and hidden Markov models and is able to detect efficiently denial of service attacks. [93] presents another system based upon system calls and Markov models to detect intrusions. This system analyzes the arguments of the system calls but is oblivious of the system call sequence. System call sequence and deterministic automata have been used in [85] to detect anomalies, which are detected when system call sequences differ from an execution trace known to be good.

4.6 Conclusion and Future Work

In this paper, we have discussed the current limitations of the semantics of the security models for mobile applications. To this end, we have presented a probabilistic version of the Security-by-Contract with Trust, which is able to guarantee probabilistic requirements. We have discussed the advantages in terms of expressiveness achieved including probability in the $S \times C \times T$ framework. Finally, we have shown a possible use-case, including $S \times C \times T$ in the Android operative system, showing the feasibility of the proposed approach.

Future extensions to this work will be the definition of probabilistic formalisms and languages, which should be used to programmatically define probabilistic contracts and policies, then to verify their compliance. This languages should be equivalent in expressiveness to the probabilistic automata that we have used to express policies and contracts. Furthermore, we are going to include the presented framework in real mobile devices, investigating if it is possible to distribute it as common mobile application, which can give users a way to better control their mobile devices.

5

MADAM: Multi-Level Anomaly Detector for Android Malware

5.1 Introduction

At the end of 2014, the number of active mobile devices worldwide was almost 7 billions and in developed nations the ratio between mobile devices and people is estimated as 120.8%. In particular, 95% of these devices are *smartphones* or *tablets* [6]. Given their large distribution, and also their capabilities, in the last two years mobile devices have became the main target for attackers [10]. *Android*, the open source operative system (OS) introduced by Google, has currently the largest market share [6], which is greater than 80%. For these reasons, Android is almost the only target of attacks against mobile devices (98,5 %) with more than 1 million of malicious applications (*apps*) available in the wild [37].

Malicious apps constitute the main vector for security attacks against mobile devices. Disguised as normal and useful apps, they hide treacherous code which performs actions in the background that threatens the user privacy, the device integrity or even user's money. Some common examples of attacks performed by Android malicious apps [135] are stealing contacts, login credentials, text messages, or maliciously subscribing the user to costly premium services. Furthermore, all these misbehaviors can be performed on Android devices without that the user can notice them or before it is too late. It has been recently reported[1] that almost 60% of existing malware send stealthy premium-rate SMS messages. One of the main reasons is that Android malware exploits operations that are not malicious by themselves but if contextualized can be used to give a revenue for the attacker. For example, the action of sending a text message is a normal operation; however, sending a text message to a premium service number *and* deleting it

[1] `http://goo.gl/dMgpxm`

from the messages outbox (to pass unnoticed) is very likely a misbehavior. These apps are part of a category of apps called *trojanized*, a subclass of *repackaged apps*, and can be found in online marketplaces not controlled by Google. However, also *Google Play*, the official market for Android applications hosted apps which have been found to be malicious[2]. Even if these apps are removed both from the market, and remotely from devices where the app has been deployed, they may have already performed their misbehavior.

In an attempt to limit the set of (dangerous) operations that an app can perform, Android has introduced its native security mechanisms in the form of *permissions* and apps *isolation*. The Android permission system enforces access control on security critical resources and operations, forcing the developer to declare beforehand the resources and operations that the app needs to access. If an app attempts to access a resource, or perform an operation without declaring the needed permission, the access is denied. For what concerns app isolation, Android exploits the underlying Linux kernel and the virtual machines in which each app is executed to ensure that an app cannot interfere with the execution of another one. Moreover, each app has its own private memory space, since every virtual machine receives a different Linux user ID and the corresponding home folder accessible only to that specific app. However, both permissions and isolation have shown weaknesses [71], which have been exploited by malicious developers.

As a motivating example, let us consider the gaming app `Baseball Superstar`[3]. While the app on the official store does not bring any harm to the device, on some unofficial channel it is possible to find a trojanized version of this app [135][55]. The only hint on the possible danger brought by this app is that the list of declared permissions includes the request to send text messages (SMS). This inconsistency is likely to pass unnoticed (according to [71]), and hence, once granted the authorization, no more evidence of the app misbehaviors are shown to the user. In fact, the app looks like a video game correctly working. However, in background, the malicious code of the malware `Geinimi` opens a back-door for a command and control (C&C) server which can instruct the device to send text messages from remote. It is worth noting that the outgoing messages will pass unnoticed since in Android messages sent by apps are not stored in the outbox, nor notification to the user are sent. Not only sending text messages impose a cost to the user, but the attacker, with the right authorizations, can also use these messages also to send private data of the user or device (such as IMEI and IMSI code). Additional motivating examples are the apps hiding the rootkit `LeNa` [135]

[2] `http://goo.gl/IGmZhk`
[3] `http://goo.gl/FWLNWw`

which stealthily installs other apps with dangerous authorizations, which are never shown to the user. The installed apps are able to communicate between them, by exploiting the Android standard communication mechanism and eluding the isolation mechanisms. Through this scheme, even a single malicious app can enable an attacker to fully control the device, and all the stored information. Moreover, these frameworks generally requires a custom operative system [67]. Along with the vast increase of Android malware, several security solutions have been proposed, spanning from static or dynamic analysis of apps [112] [106], to applying security policies enforcing data security[31] [25], to run-time enforcement [137] [67]. However, the presented solutions still present important flaws. In particular they are attack-specific, i.e. they focus and tackle a single kind of security attack, e.g. privacy leaking [137] [67], or privilege escalation (jail-breaking) [31] [32].

In this paper we present *MADAM* (Multi-level Anomaly Detector for Android Malware), a host-based Intrusion Detection System (IDS) for Android device. MADAM security enforcement is multi-layer in the sense that it spans from kernel-level to user level, to *prevent*, *detect* and *block* malicious behaviors and their causes. MADAM has been designed to be usable, i.e. with a limited overhead (1.4% of performance overhead and 4% of battery depletion) and requires almost none user interaction. MADAM is also adaptive, since it is able to progressively learn the user and device behavioral patterns, by means of a classifier (K-NN) which can continuously add new elements to its knowledge base. MADAM has been tested against two large dataset of malicious apps [135] [1] and proved to be effective in detecting and stopping malicious behaviors related to privacy leaking, stealing money through text messages and mining device integrity.

MADAM comes as an Android app that, once installed, constantly monitors the device with the following actions: (i) it assesses the risk of newly installed apps, before they are executed, by analyzing the requested permissions and reputation metadata, such as user scores and download number; (ii) it alerts the user when a newly installed app is potentially dangerous, by handling the removal process if the user does not decide to install it anyway; (iii) it monitors the amount of system calls globally issued on the device and the user activity, learning through a classifier user and device behavioral pattern and alert anomalies; (iv) it implements dynamic analysis of apps behavior with interception and blocking of potentially dangerous actions (e.g. sending text messages); (v) it enforces security rules (heuristics) which take in account device, user and app behavior.

The novel contributions of this paper are the following:

- This paper describes in details MADAM, a light-weight multi-level anomaly detector for Android malware.

- It is proposed a classification of Android malware in 8 macro-groups, to better divide the several malware families which can be found in the wild.
- The paper details a methodology to detect global misbehaviors exploiting a white list-based analysis of system events, intercepted at three different levels (i.e. system calls, app and user).
- A configurable and adaptive heuristics-based system to detect and stop app dangerous behaviors is presented.
- It is presented a module that performs a static pre-filtering based on app metadata, such as the declared permissions, to find apps that are considered as risky. If this is the case, then MADAM focuses its dynamic analysis on them. It then presents the advantages for performance and accuracy given by the inclusion in the MADAM framework of this component to classify Android apps at deploy-time.
- A thorough analysis on the MADAM classification capabilities is reported showing the performance of the chosen classifier and the detection results.
- The results of MADAM analysis against two dataset of malicious apps are reported. MADAM reported a detection rate of 95% on a testbed of 1300 malicious apps.
- A study on usability to assess the MADAM overhead has been conducted. MADAM shows a low performance overhead (1.4%), low battery depletion (4%, 1 hour on 24 hours of standby) and low false positive rate (one false alarm per day, in average).

The rest of the paper is organized as follows: Section 5.2 describes the relevant works in related field. Section 5.3 reports background information on Android security and threats, also proposing a malware classification in 8 macro-groups. Section 5.4 presents the MADAM framework describing in details the components and the workflow. Section 5.5 reports the analysis on the performance of difference classifier, motivating the choice of the K-NN. Afterward reports the detection results on two large dataset of malicious applications, namely Genome and Contagio Mobile, together with the performance and false alarm rate analysis. Section 4.6 concludes presenting possible framework extension.

5.2 Related Work

Crowdroid [34] is a machine learning-based framework that recognizes Trojan-like malware on Android smartphones, by analyzing the number of times each system call has been issued by an app during the execution of an action that requires user

interaction. A genuine app differs from its trojanized version, since it issues different types and a different number of system calls. Crowdroid builds a vector of m features (the Android system calls). Differently from this approach, MADAM uses a global-monitoring approach that is able to detect malware contained in unknown apps, i.e. not previously classified. Furthermore, on *Crowdroid* only two trojanised apps have been tested, whereas on MADAM we tested ten real malware. A similar approach is presented in [47], which also considers the system call parameters to discern between normal system calls and malicious ones.

Another IDS that relies on machine learning techniques is *Andromaly* [14], which monitors both the smartphone and user's behaviors by observing several parameters, spanning from sensors activities to CPU usage. 88 features are used to describe these behaviors; the features are then pre-processed by feature selection algorithms. The authors developed four malicious apps to evaluate the ability to detect anomalies. Compared to Andromaly, MADAM uses a smaller number of features (13), and has been tested on real malware found in the wild, and shows better performance in terms of detection and, especially, of false positives rate. After the learning phase, the false positive rate of MADAM is 0.0001, whereas that of [14], which uses a sampling method similar to that of MADAM and with a comparable sampling rate (2 seconds), is 0.12. The detection rate of MADAM is 93%, while that of [14] is 80%.

Other approaches only monitor misbehaviors on a limited number of functionalities such as outgoing/incoming traffic [46], SMS, Bluetooth and IM [12], or power consumption [75] and, therefore, their detection accuracy is higher of other work but less general. [113] monitors smartphones to extract features that can be used in a machine learning algorithm to detect anomalies. The framework includes a monitoring client, a *Remote Anomaly Detection System* (RADS) and a visualization component. RADS is a web service that receives, from the monitoring client, the monitored features and exploits this information, stored in a database, to implement a machine learning algorithm. In MADAM, the detection is performed locally and, more importantly, in real-time. [128] proposes a behavior-based malware detection system (*pBMDS*) that correlates user's inputs with system calls to detect anomalous activities related to SMS/MMS sending. MADAM is more general since it considers all the activities on a smartphone. A further framework targeted at SMS/MMS monitoring is *Proactive Group Behavior Containment* [30], which is aimed at containing malicious software spreading in these messaging networks.

[69] and [100] propose *Kirin* security service for Android, which performs lightweight certification of apps to mitigate malware at install time. Kirin certification uses security rules that match undesirable properties in security configuration

bundled with apps. [112] performs static analysis on the executables to extract functions calls usage using `readelf` command. Hence, these calls are compared with malware executables for classification.

5.3 Background

5.3.1 Mobile Malware

Due to the predominance of the Android OS, more than 98% of current malware for mobile device is target for Android [37]. These pieces of malware are generally diffused through *unofficial* app markets, which distribute repackaged free versions of apps that have a cost on the official market Google Play. *Repackaged apps* are the main vector of malware for Android devices. These app include both the normal code of the app plus malicious code that runs in background performing misbehaviors. A subclass of this class is that of *trojanized apps*, which perform extremely dangerous actions.

According to Kaspersky [37], more than ten million malicious apps for Android were available at the end of 2013. More recently, a report for the first half of 2014 [90] presents 20 types of new malware. Notwithstanding the huge number of malicious apps and threats, Android malware can be divided in families of malware that show exactly the same behavior. In 2012 the Android Malware Genome Project [135] presented a collection of more than 1000 thousand malicious apps divided in 49 families. A more updated classification of Android malware [116], refers to a website[4] that lists 154 malware families. During the analysis of the malware, we have noticed that, though the number of malware families is relatively high, the classes of threats brought by these malicious apps is quite limited in number. In fact, by only taking into account the goal of the malware, we have seen that misbehaviors performed by malware can be grouped in the following 10 categories:

- **botnet functionality**: open a backdoor on the device, waiting for commands which can arrive from an external server or an SMS message;
- **gains root access**: perform buffer overflow to get super user privileges on the device;
- **SMS trojan**: send SMS messages stealthily and without the user consent, generally to subscribe the user to a premium services, or send spam messages to all og user contacts;

[4] http://forensics.spreitzenbarth.de/

- **steals location information**: take pieces of data from location interfaces and sends them to an external server without the user implicit or explicit consent;
- **steals private information**: superset of the previous group which also attempt to steal device IMEI and IMSI, contacts, message inbox or social network account data;
- **installs other apps or binaries**: install apps with new authorizations to increase the capability of harming the system.
- **banking trojan**: exploits authentication mechanism of some bank institutes, based on SMS messages to authorize unwanted transactions;
- **infects a connected computer**: send malicious payload on a personal computer when the infected mobile device is connected via USB;
- **make the device unusable**: continuously shows an activity on top of the screen preventing the user from interacting with the mobile device;
- **violate file integrity**: modify or delete data from the device without the user consent.

We can furtherly refine this clustering by merging the two classes “Banking Trojan” with “SMS Trojan” in a single class “Trojan”, since banking Trojans cleverly exploits SMS messages to perform unsolicited bank transactions. Moreover, we can also merge “Steals location information” with “Steals Private Information” in a single misbehavior class, since location is a private data itself. Considering that the various pieces of malware in the wild perform one or more of the aforementioned misbehavior, we propose the following malware classification:

(i) *botnet*: malware performing the “botnet functionality” misbehavior;
(ii) *rootkit*: malware performing the “gains root access” misbehavior;
(iii) *SMS trojan*: malware performing “SMS trojan” or ”banking trojan” misbehavior;
(iv) *spyware*: malware stealing private or location information;
(v) *installer*: malware installing additional apps without the user consent;
(vi) *vessel*: malware that waits for the infected device to be connected via USB to a personal computer to be infected.
(vii) *ransomware*: malware that makes the device unusable and pretends to make it again usable after a ransom is paid.
(viii) *trojan*: malware with generic misbehavior not belonging to the former categories.

Some malicious apps fall at the same time in more than one of these categories. For example, several rootkit also hide inside botnet functionality, or capacity to install apps and sending SMS messages.

It is worth noting that malware for Android usually perform malicious actions whose effects are indeed dangerous, and that may inflict even a direct monetary loss to the user. However, these are standard actions and, for this reason, misbehaviors of malicious apps are really unlikely to be discovered by users on time. In fact, malware prefer subtlety to aggressiveness. Moreover, malicious apps do not follow a fixed scheme in the permission request. Some malicious apps require dangerous permissions, which clearly show the potential malicious intention of the malware, whereas others cleverly hide the malicious permissions with legitimate ones. As an example, a trojanized app for instant messaging can ask for the `SEND_SMS` permission legitimately and then exploit it maliciously. Finally, some pieces of malware do not ask for permissions since they acquire the root privileges at the kernel level, avoiding the permission security mechanism. This motivates the design of a security framework which automatically detects apps' malicious behavior, analyzing elements (features) non-observable by the user.

5.3.2 Android Security

The Android OS includes native security mechanisms to protect the user and the device from malicious apps. These are based upon *Access Control* and *App Isolation*. Access control exploits the *permission system* to protect access to security critical resources and operations. In particular, if an app wants to perform a security critical operation, e.g. sending a text message or wants to access a security critical resource, e.g. contacts in device contact list, the developer has to declare this intention requesting the permission beforehand in the `AndroidManifest.xml` file, part of every Android app. Thus, the developer declares all resources and operation performed by the app through permissions. Then, when the user is installing the app (at deploy-time) the permission of the manifest file are shown and the user decides if she wants to grant such authorizations or not. Note that the user can only accept to authorize the app with all the permissions or deny them all. . After several critics to this too corarse-grained "all or nothing" approach [57], Android introduced the possibility to grant and revoke single permission to installed apps through the system app `AppOps`. However, the improvement brought by `AppOps` is quite limited. In fact, whenever an app tries to perform an operation without the permission being granted, the Android *permission checker* will deny the operation at run-time, returning an exception that, if not handled, causes the crash of the app.

Android relies also on *app isolation* to improve the security of apps. Every Android app runs in a different virtual machine (Dalvik Virtual Machine, DVM), by

seeing memory as if it is the only app running on the device. Moreover, every DVM is assigned with a separate Unix user ID. Thus, any app also has its private storage space (home folder) not accessible to any other app or user. Notwithstanding, Android app can communicate between them using *Intents*, which are a communication mechanism provided and handled by the Android OS, or through inter process communication (IPC). Maliciously exploiting these communication mechanisms, it is possible to elude the app isolation security mechanism, performing attacks like the *Confused Deputy Attack* [31].

5.3.3 Characterization of the Problem

Android malware are distributed in the form of Trojanized apps, which perform malicious behaviors in background and, hence, are difficult to detect. Current anti-virus software is based on a static analysis and a black-list approach aimed at finding known malicious patterns in the app code or binary. However, the amount of malware samples available grows every day [90]. These new pieces of malware come either as modification of existing samples or totally new malware pieces and are called Zero-Day threats. By definition, Zero-Day threats defy any black-list approach, since their code (or binary) will be likely not registered in any database. The procedure of finding the new threat and updating anti-virus software on user devices can also take several days, in which the user remains vulnerable to these attacks.

On the other hand, the weakness of the permission system is that the decision on installation is responsibility of the user only. In general we can say that permissions effectiveness is extremely limited because several users do not understand them [71, 70] and often they simply install the app without reading the requested permissions, making useless the whole security mechanism. For these reasons, we need a system that checks for suspicious activities, in an automatic way, without relying on the users.

5.4 MADAM

The Multi-Level Anomaly Detector for Android Malware (MADAM) is an host-based IDS for Android devices. MADAM monitors multi-level features, namely system call, API calls and user activity. The system calls describe the device behavior at kernel level, in fact, any action performed by applications or OS is translated in a sequence of system calls. For this reason the system call analysis has already

been used in previous work to detect intrusions [47]. On the other hand, monitoring specific API calls allows to control which high level operations are performed on the device by the OS or applications, e.g. sending a text message, start a phone call, start an application etc. Finally monitoring the user activity means to understand when she is interacting with the phone and when she is not, relating these two profiles with the device activity. In fact, when the user is interacting with the phone, the number of events (both at system call and API level) monitored on the device is generally higher than when the user is not interacting. Through these features MADAM is able to discern between normal and anomalous device behavior and to find the malicious actions performed by apps. In particular, is considered as normal the behavior of a device non-infected by any malware, learned by MADAM in a training phase described in the following. The behaviors which strongly differs from the normal ones are considered anomalous and deemed by MADAM as malicious. Due to the anomaly-based approach, MADAM targets all the aforementioned malware classes. Other main elements analyzed by MADAM are the permissions declared by apps and apps' reputation metadata, such as rating, marketplace and download number. These elements are used by MADAM for performing a risk assessment of the app at deploy-time. MADAM is aimed at detecting those app misbehaviors which otherwise are not observable to users and Android native security mechanism. To this end it combines the results of the static analysis of app with the multi-level monitoring of the device behavior. To effectively enforce security, MADAM also implements heuristics aimed at stopping known malicious actions (black-list). Examples of used heuristics are "prevents apps from sending more than one SMS to a number not in the address book" or "kill an app execution if it creates an amount of processes greater than a specific threshold". In the following we will present the MADAM architecture, detailing the various components and describing their interaction.

5.4.1 Main Components and Workflow

MADAM is composed of four main functional blocks which analyze the smartphone activities at different levels, as shown in Fig. 5.1. The first one is the *App Classification Module* (App Evaluator), which executes an app pre-filtering by analyzing static features of the app package (`apk`) before the app is installed on the device, eventually populating a set of suspicious apps. The second block is the *Global Monitor*, which monitors the device and OS features at three different levels, i.e. kernel (system call), API and user, to continuously shape the current behavior of the device itself (i.e., not of any particular app, but of the system as a whole), by

classifying it as *genuine* (normal) or *malicious* (anomalous). The third block is the *Per App Monitor*, which implements a set of heuristics to monitor the actions performed by the set of suspicious apps generated by the AppClassifier, and stops malicious actions and then handles the procedure for removing malicious apps. Finally, the *user interface* handles notifications to device user.

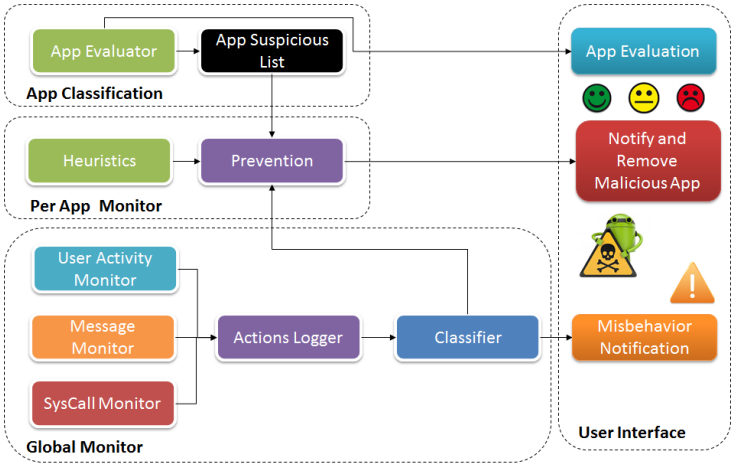


Figure 5.1: Architecture of MADAM

Figure 5.2 reports the workflow of the MADAM framework. The flowchart on the left describes the pre-filtering procedure, which stars at the deployment of a new app. The app evaluator will analyze the application package looking for required authorizations and other metadata (detailed in Sect. 5.4.3) to assess the app risk. If the app is deemed as malicious is added to a list of suspicious apps. The app in the suspicious list are subject to the heuristics of the per-app monitor. Actions performed by app deemed as genuine will be stopped only if specifically violating one or more heuristics. This allows to reduce the amount of false alarms. The right workflow of Figure 5.2 describes the behavior of MADAM when an alarm is raised by the global monitor. MADAM checks if there are apps in the suspicious list. If the suspicious list is empty, the alarm will be recorded for subsequent user analysis. As discussed in the following, this event is unlikely. If some apps are in the suspicious list, MADAM verifies if an heuristic has been violated by one of the app in the list. The per app monitor stops the violating actions before they take

place, through an hijacking mechanism detailed in Section 5.4. Afterward MADAM proposes the app for removal, leaving the final decision to the user.

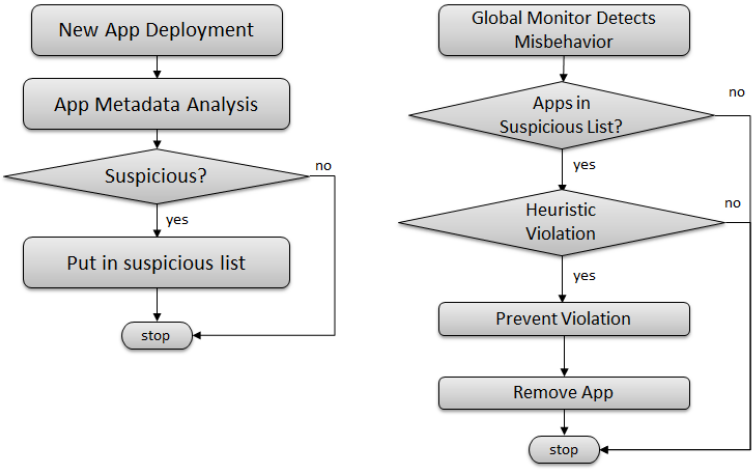


Figure 5.2: MADAM Workflow

5.4.2 Global Monitoring

The global monitoring component of MADAM is based on two proximity-based (K-NN) *Classifiers*, whose task is to detect anomalous behaviors. As discussed, a behavior is described by a set of *features*, collected at run-time and representative of the current device activity. The features are extracted from the different kinds of dynamic events (refer to Figure 5.1): *User Activity*, *API Level* (in particular, *Text Messages*) and *System Call* (SysCall).

The global monitoring component includes two cooperating classifiers. The first instance of the global classifier is a short-term monitor (classifier) with T_{short} sec, whereas the second instance constitutes a long-term monitor with T_{long} sec (both values are configurable at run-time). The cooperation of these two instances detects different types of misbehaviors. The short-term monitor is more effective in detecting “spiky” misbehaviors, i.e. with sudden, brief and sharp increase of the system call occurrences. On the other hand, the long-term monitor is aimed at detecting misbehaviors that distribute their action constantly in a long period of time, such as spyware, i.e. whose effect is not immediate.

The usage of computational intelligence (classifiers) and statistical techniques for intrusion detection is a well known approach [47], which have already been exploited on other OSes and for network analysis[125]. In fact, regardless of the specific environment, intrusion detection can be modeled as a problem of *binary classification*. The binary classification problem for intrusion detection can be formally defined as follows. Let us consider the set of classes $\Omega = \omega_0, \omega_1$ where ω_0 is the class of good behaviors and ω_1 is the class of malicious behaviors. Then, given a behavior $B \in \mathbb{R}^m$, $B = \{f_1, \ldots, f_m\}$, where m is the total number of features describing the behavior, assign to B the correct class in Ω, through a function $D : \mathbb{R}^m \rightarrow \Omega$ named classifier.

Several classifiers correspond to this description and can be used to solve the classification problem. Among them, the $k - NN$ (k - nearest neighbor) classifier [118] is the one that best meets the MADAM needs for the following reasons. (i) All the m features describing a behavior in MADAM are numerical, as explained in the following.(ii) The rationale of MADAM's classification process is that an intrusion represents a consistent deviation from the device normal behaviors and using numerical features it is possible to geometrically represent this difference. The k-NN classifier exploits this geometric representation to classify behaviors close to genuine ones as belonging to class ω_0 and the behaviors close to the malicious ones as belonging to class ω_1. Formally stated, given a new element (behavior) x and considering y be an element whose class is already known, the k-NN computes the similarity between x and y through the following relation:

$$Similarity(x, y) = -\sqrt{\sum_{i=1}^{m}(x_i - y_i)^2}$$

where x_i and y_i are the features of the vectors x and y. Then assigns to x the class of the y which yields the highest value for this similarity.

The global monitor's classifier (Figure 5.1) is trained to recognize a list (white list) of real genuine behaviors collected on devices. As discussed, MADAM is an anomaly based IDS, which by definition, alerts as anomalies the behaviors which appreciably differs from the known ones. However, if a classifier is only trained to recognize genuine behaviors, it will never report any alert. For this reason, the classifier is also trained with a set of synthetic behaviors which are appreciably different from the set of genuine ones, but not related to a specific malware. Thus, no malware is detected by the global monitor because its signature is known by MADAM. Hence, MADAM should be effective against zero day attacks, given that they performs behavior which differ from the known ones. The generation of syn-

thetic behaviors will be detailed in Section 5.5. In MADAM, each behavior vector B_i is composed by $m = 14$ features. Each of the first eleven features records the amount of the 11 different system calls issued in the amount of time T_k. The complete list of monitored system calls is reported in Table 5.1: these are the syscalls related to file operations and network access and are relevant because the greatest amount of operations of Android are translated at a low level as editing of system files. The 12-th feature represents the user activity (idleness) and is defined as follows: at each time interval T_k.

$$f_{11} = \begin{cases} 1 & \text{if screen is off and user is not calling.} \\ 0 & \text{otherwise.} \end{cases}$$

We note that user activity is strongly related to the overall activity of the phone. We can categorize the general status of the phone in two states. In the first state either the user is actively interacting with the phone and the screen is on, or the screen is off but a phone call is ongoing, otherwise, in the second state, the phone is not active. In fact, when the user is active, the phone has to show interactive contents on the screen and receives inputs from the user, or handles the elements involved in a phone call. In these cases, a large amount of system calls is generated. On the contrary a low number of system call is generated. Hence, it is possible to delineate two different profiles, as shown in Table 5.1, where the 12-th feature represents whether the user is active (1) or not (0). Considering that 90% of available Android malware are related to unsolicited outgoing text messages, i.e. SMS trojan [135], the amount of outgoing text messages for every T_k interval is also used as a feature (f_{13}). Finally, the feature f_{14} represents the amount of text messages sent to a recipient which is not in the device contact list.

open	ioctl	brk	read	write	exit	close	sendto	sendmsg	recvfrom	recvmsg	idleness	SMS Num	SMS susp
6	19	18	1	4	0	7	16	2	2	0	0	0	0
147	652	192	711	4	282	229	7	15	7	13	1	0	0

Table 5.1: Comparison of Behaviors: User Idle (Top) vs User Active (Bottom).

5.4.3 Metadata Analysis of Installed Apps

When a new app is installed on the device (deploy-time), MADAM intercepts and hijacks the installation event through the MAETROID method, presented in Chapter 2. MADAM analyzes the metadata of the new app package to assess its risk. The analysis exploits five parameters, namely: (i) the permission declared in the

manifest, (ii) the market of provenance, (iii) the total number of downloads, (iv) the developer reputation and (v) the user rating. MADAM automatically extracts all these information in a process which is totally transparent to the user. The user can decide whether she prefers to receive a notification of the App-Evaluator decision (Figure 5.3), or to keep the process invisible. The five parameters are analyzed through a hierarchical algorithm [55], which returns a decision on the app classifying it as *safe* or *risky*. Based on this decision, the user can choose whether to remove the new app. If the user chooses to install a risky app (or decides not to receive notifications) the app package name is recorded in the MADAM list of suspicious apps, as formerly discussed. In the following, we assume that the user chooses the transparent approach, allowing the other components of MADAM to enforce security on the device.

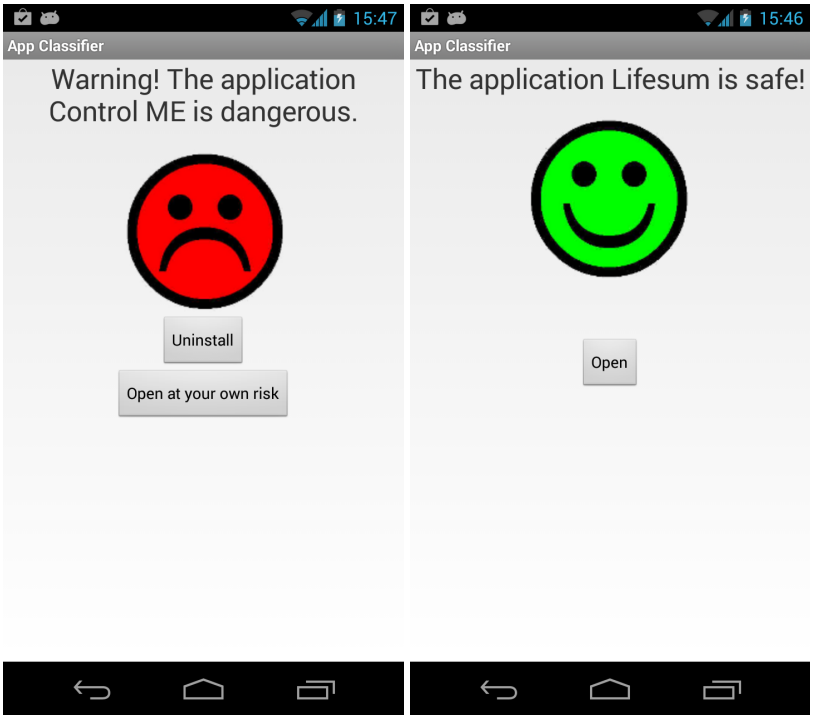


Figure 5.3: App Evaluator Decisions Shown to the User for Safe (left) and Risky (right)

5.4.4 Intercepting System Calls

To intercept (i.e., hook) system calls on Android, MADAM does not require a modified kernel or custom operative system. In fact, MADAM exploits a kernel module to hook those system calls that are considered critical from the point of view of security. The kernel module is loaded through the `insmod` command and interacts with the rest of the MADAM framework through a shared buffer, used to exchange information between the application level and the kernel level. Loading the kernel module requires the usage of the `insmod` command, which requires the super user privileges to be taken. Hence, though there is no need for a custom operative system or kernel, MADAM requires the device to be rooted (jail-broken).

5.4.5 Exploiting the XPosed Framework

The MADAM prevention module exploits the *XPosed Framework* [35] to intercept calls to security relevant API functions. The XPosed Framework is based on an app that modifies the `/system/bin/app_process` executable, to make it load a `JAR` file at device startup. The `JAR` file changes the references to the default API calls, allowing the redefinition of any method of every class. Therefore, Android will call henceforth the redefined method instead of the predefined one. Through Xposed framework, MADAM redefines the security relevant API, such as `SendTextMessage()` (method to send SMS messages), reading the method's actual parameters and the issuing app. Furthermore, MADAM is able to avoid that the method is executed, preventing the negative effect of the misbehavior.

5.4.6 Heuristics and Prevention

The decision whether a performed operation has be stopped is demanded to a module the implements a set of heuristics which is configurable and extensible. Heuristics apply to both global events (like outgoing text messages) and to events generated by those apps which have been considered suspicious by the App Evaluator module. MADAM proposes two sets of heuristics: (i) *System Call Heuristics*. These heuristics have been created by analyzing a set of app behaviors considered as malicious, using a tree classifier to infer the common elements of these behaviors. From this analysis we have seen that misbehaviors are often related to sudden increase of some system calls, namely `open, read, write, close`. Given these system calls, it is possible to find the app which caused the sudden increase in the list of suspicious ones, monitoring the system calls generated by the single app and deem it as malicious. (ii) *API Level Heuristics*. These heuristics

represent high-level behaviors that are considered suspicious, which have been inferred from known malware misbehaviors. Before introducing these heuristics we define some preliminary concepts. Android devices have a *messaging default app*, which is the app that by default handles the operations related to text messages. Unless changed by user, the messaging default app is the native one. Other apps may send text messages, asking for the permission `SEND_SMS`. However, Android does not introduce by itself controls on the message recipient and on the number of sent messages and, in fact, SMS Trojans exploit these vulnerabilities. Therefore, the heuristics introduced by MADAM are aimed at mitigating these threats. Namely the API level heuristics that we are considering are the following:

- Block text messages that are sent by a non-default message app. If this heuristic is enabled, only the messages actively written and sent by the user, through the messaging default app, will be allowed. MADAM will stop any other message sent by other apps. Furthermore, if the message is sent by an app included in the suspicious list, at the act of sending a message, the app will be proposed for removal.
- Block text messages sent to numbers that are not in the user contact list. When an app, different from the default messaging app, attempts to send a text message, MADAM hijacks the action and verifies if the recipient is in the contact list. If the recipient is not in the contact list, the action is considered a misbehavior and is put on hold. If the sending app is in the list of the suspicious app, and the global monitor alerts a misbehavior, the app is proposed for removal. If the user stops the process of app removal, or if the app is not in the suspicious list, the user is asked if she wants to allow the message to be sent.
- Limit the amount of outgoing message per period of time. This heuristics allows the user to set a maximum number of messages that can be sent by an app per period of time. Both the time and the number of messages are configurable. Outgoing messages are allowed till the threshold is reached, otherwise the action is forbidden. Moreover, if the global monitor notifies a misbehavior, and the sending app is in the suspicious list, the app is proposed for removal.
- Limit the amount of processes generated per app. The user can set the amount of processes that an app can fork. If the number of processes overcomes the threshold, the app is killed along with all the generated processes. Furthermore, if an alert has been issued by the global monitor, and the stopped app is in the suspicious list, the app is proposed for removal.

Both API level and system call heuristics can be activated or deactivated manually by the user, as to increase or reduce the severity of the prevention system.

5.5 Results

This section presents the experimental results of the MADAM framework. First of all, we recap some preliminary notions used for evaluate the performance indexes. Then, we present a first a set of experiments to assess the classifier accuracy, performed through cross validation (K-Fold). Afterward, a second set of experiments evaluates the MADAM accuracy in real usage contexts, focusing on the detection rate against malware coming from two large databases. A further set of experiments has been conducted to assess the amount of false alarms. Finally, the impact of MADAM on performance and energy consumption has also been evaluated through benchmarking applications.

5.5.1 Evaluation Indexes

In a binary classification process elements are divided in four sets that, in the context of malware detection, can be described as follows:

- TP: True Positives, which are the events of an attack correctly classified;
- TN: True Negatives, which are normal events correctly classified as non-attack;
- FP: False Positives, which are normal events classified by mistake as attacks;
- FN: False Negatives, which are attacks that are not recognized by the IDS.

A typical representation for these four sets is the binary *confusion matrix*, which is depicted in Table 5.2.

Table 5.2: Binary Confusion Matrix

	0	1
0	TN	FP
1	FN	TP

The confusion matrix reports the elements correctly classified on its diagonal, whilst all the other elements are errors. Moreover, from the confusion matrix, the following indexes can be easily computed. The first is the *accuracy*, which is defined as the number of correctly classified elements divided by the total number of classified elements. The accuracy is the index of the classifier goodness, and is defined as:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Other two important indexes are the *False Positive Rate* (FPR) or False Alarm Rate, and the *False Negative Rate* (FNR):

$$FPR = \frac{FP}{FP + TN} \qquad FNR = \frac{FN}{FN + TP}$$

These two indexes estimate the classification error and are meaningful in understanding the effectiveness. In the particular case of IDS, the FPR is also important to estimate the usability. In fact, an IDS showing an excessive FPR can bother the user who will likely deactivate or remove the IDS.

5.5.2 Classifier Performances

The classifier exploited by MADAM is the k-Nearest Neighbor (k-NN), which is a similarity-based classifier, i.e. it classifies two similar elements as belonging to the same class. The similarity measure used by the k-NN classifier is the euclidean distance measured in the features space, i.e. two elements are considered similar if geometrically close in the features space [118], [86]. In a binary classification process, where the labels of elements y_i are in the set $\mathcal{Y} = \{0, 1\}$, the output of the k-NN classifier for a new element x' is computed using the function f_{KNN}:

$$f_{KNN}(x') = \begin{cases} 1 & \text{if} \quad \Sigma_{i \in \mathcal{N}_K(x')} y_i \geq 0 \\ 0 & \text{if} \quad \Sigma_{i \in \mathcal{N}_K(x')} y_i \leq 0 \end{cases}$$

where $\mathcal{N}_k(x')$ is the set of the K nearest neighbors of the element x'.

The similarity-based approach of the k-NN classifier is well suited with the rationale of the classification problem of MADAM. In fact, MADAM is trained with known behaviors, representing genuine device activities (white list), and artificially generated malicious behaviors, which are strongly different (distant) from the known ones. New behaviors that are similar to the known genuine behaviors are considered benign as well. On the other hand, unknown behaviors which are closer to the artificial behaviors, are classified as malicious. This choice is consistent with the anomaly-based (white list) approach, which aims at detecting behaviors that strongly deviates from known good ones, instead of searching those behaviors close to the ones known as malicious.

To further justify our classifier selection, we have considered other classifiers used for numerical (quantitative) features, namely Linear Discriminant Classifier (LDC), Quadratic Discriminant Classifier (QDC), Multi-Layer-Perceptron with backpropagation (MLP), Parzen Classifier (PARZC) and Radial Basis Function (RBF). All the classifiers have been trained with the same datasets and the same validation technique. The k-NN classifier gives the best classification results among all

other classifiers, achieving the max accuracy when $k = 1$. Details on the experiments are discussed in the following.

Testbed

Classification tests have been performed on two datasets of unique elements (vectors) representing behaviors collected through a short term monitor ($T_{short} = 1s$) and a long term one ($T_{long} = 60s$). Each dataset contains both genuine vectors and anomalous ones. The genuine vectors are behaviors that have been collected on a real Android device (`Samsung Galaxy Nexus`). The behaviors have been generated in different activity conditions, i.e. screen-off native status, user interaction with native status, screen off heavy load, user interaction heavy load, user playing heavy load. Instead, malicious behaviors have been created artificially to reproduce situations strongly different from the ones reported in the genuine behaviors, e.g. by creating vectors with a high number of issued system calls but a low user activity.

The first dataset, called henceforth *short testbed*, is composed of 830 vectors with 14 features. This dataset includes behaviors (system calls and user activity) collected in $T_{short} = 1s$. The dataset is divided in 747 genuine elements and 83 malicious elements. The second dataset, called *long testbed*, is composed of 780 vectors with 15 features (system calls, user activity, outgoing messages), divided in 656 genuine vectors and 124 malicious ones.

Experiment Description

We have tested the accuracy of six different classifiers on the datasets using hold-out (i.e. division in training and testing set), with the K-Fold Cross Validation approach, with $K = 5$ (i.e., the entire dataset is used both as training and testing set in K iterations, whose results are averaged). The classification results are reported in table 5.4 and 5.3. In detail, each table reports, for each classifier, the total amount of false positives (FP) and false negatives (FN) issued in the $k = 5$ experiments of the K-Fold cross validation. FPR and FNR instead are averaged on the 5 experiments. Finally, the global accuracy (averaged) is reported for any classifier. Details are not reported for classifiers performing with an accuracy lesser than 50%. As shown in the tables, the K-NN gives the best accuracy results for both datasets. In the short term experiments, both LDC and K-NN yield the same accuracy, but the FNR (real alarms passing unnoticed) is much larger (more than 18%) in the case of LDC. Hence, the aim of keeping a “safe-side” approach motivates our preference on the K-NN also for the short term classifier.

Table 5.3: Classification Results on Short Term data.

Classifier	FN	FP	FPR	FNR	Accuracy
K-NN	11	11	1.47%	12,2%	97,35%
LDC	16	6	0.8%	17.78 %	97.35%
QDC	N/A	N/A	N/A	N/A	$< 50\%$
MLP	N/A	N/A	N/A	N/A	$< 50\%$
PARZC	19	14	9,3%	21%	96%
RBF	16	20	2.6%	17.8%	96,7%

Table 5.4: Classification Results on Long Term data.

Classifier	FN	FP	FPR	FNR	Accuracy
K-NN	10	16	1.9%	8%	97,7%
LDC	33	6	0.9%	26,4 %	95%
QDC	17	9	1,4	13,6%	96,6%
MLP	N/A	N/A	N/A	N/A	$< 50\%$
PARZC	N/A	N/A	N/A	N/A	$< 50\%$
RBF	21	19	2.6%	16.8%	94.9%

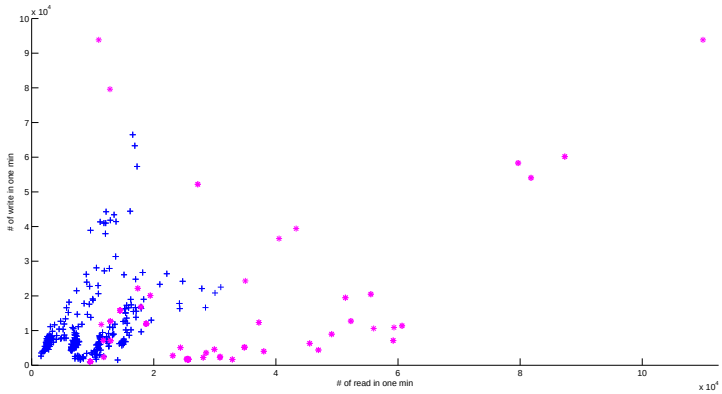


Figure 5.4: Representation of the Dataset in the `write` and `read` Feature Space.

These results are sound with the nature of the specific MADAM classification problem. In fact, genuine behaviors are not homogeneously distributed but, instead, they can be seen as separate “clouds” or clusters of similar behaviors. The reason is that, for any device, there are different usage patterns, all of which are legitimate. As an example, let us consider the differences in device activity when the screen is off and when the user is actively interacting. This difference influences

the frequency and number of the issued system calls, which is shown in Figure 5.4, where the graph reports a two-dimensional representation of the long-term dataset. The blue plus (+) represent the genuine behaviors, while the magenta stars (*) represent the malicious ones. The graph only reports, for the sake of representation, the two features representing the amount of `read` and `write` system calls issued. From the graph we can seen that the genuine behaviors form six agglomerated clusters, four in the bottom left of the graph and two with a higher number of system calls. In particular, the small cluster of genuine behaviors with more than $4 \cdot 10^4$ `write` is related to heavy device usage, i.e. gaming. It is also possible to deduce that the malicious behaviors are not easily separable from genuine ones, using those classifiers which try to use hyperplanes or hyper-curves to divide classes. For this reason, proximity-based classifiers, such as K-NN, are more suitable to address this specific classification problem.

5.5.3 Malware Detection

To verify the effectiveness of MADAM, we have extensively tested the framework against two large datasets with more than 1,300 malicious Android apps. Experiments have all been performed on a jailbroken `Samsung Galaxy Nexus` running Android stock Jelly Bean version 4.3. MADAM has been used with the following configuration: the app evaluator shows the decision for each installed app, and the global device monitor alerts for every detected misbehavior. The active heuristics are “Stop text messages sent to numbers not in contacts”, “Limit the amount of messages to 3 per minute”, “Limit the amount of processes per app to 20”.

The Genome Dataset

Genome [135] is a collection of 1,242 malicious Android apps collected in the years 2010 and 2011. The apps are divided in 49 malware families that include almost all malware categories discussed in 5.3, namely botnet, rootkit, SMS trojans, spyware, installer and trojan. The vast majority of genome apps come from Chinese unofficial marketplaces and some malware are (or were) effectively dangerous only if installed on Chinese smartphones, i.e. with the user located in China with a Chinese mobile operator. For example several apps belonging to the SMS trojans category, contact premium numbers, but without adding the national prefix. Thus, the number is not reachable from outside the country and the malware is ineffective. However, regardless of the location, if the app attempts to perform the misbehavior, MADAM is still able to detect and stop it (even if non harmful). Out of the 1,242 apps of the original dataset, MADAM could have been tested on 809

Table 5.5: MADAM Analysis of the Genome Dataset. (**P** = Pre-filtering, **R** = Runtime detection).

Malware	**Samples**	**Type**	**P**	**R**
ADRD	22	Spyware	√	×
Asroot	8	Rootkit + Installer	√	√
BaseBridge	122	SMS Trojan	√	√
BeanBot	8	SMS Trojan	√	√
Bgserv	9	SMS Trojan	√	√
DogWars	1	SMS Trojan	√	√
DroidCoupon	1	Rootkit + Installer	√	√
DroidDream	16	Rootkit + Spyware	√	√
DroidKungFu	402	Rootkit + Installer	√	√
FakePlayer	6	SMS Trojan	√	√
GamblerSMS	1	Spyware	√	×
Geinimi	69	SMS Trojan	√	√
Gone60	9	Spyware	√	√
GPSSMSSpy	6	Spyware	√	×
HyppoSMS	4	SMS Trojan	√	√
Jifake	1	SMS Trojan	√	√
Kmin	52	Spyware	√	√
NickySpy	2	Spyware	√	×
Plankton	11	Spyware	√	×
SndApps	10	Spyware	√	√
Spitmo	1	Spyware	√	√
Tapsnake	2	Spyware	√	×
Walkinwat	1	Spyware	√	√
YZHC	22	Rootkit + Installer	√	√
zHash	11	Rootkit + Spyware	√	√
Zsone	12	SMS Trojan	√	√
Total	809	···	100%	95%

apps: the apps that are not considered in this analysis are those that either “crash” during start, without performing any action, or those app awaiting commands from a C&C server (botnet), which is not active anymoreMoreover, a consistent set of malicious apps rely on OS vulnerability which have been fixed in the new Android releases. Thus, these pieces of malware are not effective anymore on the Android version that has been used for the experiments. In particular, 63 apps falls in this last category.

MADAM detection results are reported in Table 5.5. The second column from right (**P**, pre-filtering) specifies if the application has been detected by the App Evaluator module at deploy-time ($\surd$) or not ($\times$). The right column (**R**, runtime detection) specifies instead if the application has been detected at run-time, showing an alarm to the user and/or stopping the malicious behavior. The global detection rate on the 809 analyzed applications has been of 100% for the App Evaluator (**P**) and 95% for the global monitor (**R**). Malware belonging to the rootkit category are detected by the short term and long term k-NN classifiers. In fact, we have observed that they cause strong fluctuations in the number of issued system calls, not coherent with the current user activity, both if she is actively interacting with the device or not. Furthermore, all rootkits using the Rage Against The Cage [98] technique to perform buffer overflow are detected by the per app classifier, which observes a large number of forked processes, belonging to the app. Installer malware are blocked as soon as they try to install a new application. In fact, MADAM is able to detect through the short term K-NN the behavior change caused by the unsolicited installation of a new package. Also, the App Evaluator intercepts the event of installation of a new package, regardless of the installation source. In this case, MADAM notifies the user, even if the installation of a new package is issued from a malicious app. Furthermore, MADAM is able to notify the event of outgoing SMS message through the global monitor, even if the message is stealthily sent by an app.

We not that, normally, Android does not allow the monitoring of the event of outgoing text messages sent by an app, unless the app developer explicitly declares the notification intent. For this reason, MADAM intercepts, controls (extracts text and recipient) and even block any outgoing SMS. Thus, MADAM is able to detect the app which is sending SMS message, stopping the action if deemed as suspicious from the per-app classifier and global one. By joining these two actions, MADAM results to be totally effective against SMS trojans (100% of accuracy on the testbed). Spyware actions are detected through the global monitor. In particular the long term k-NN classifier detects the access attempts to device resources, not related to the user activity. Analyzing a set of 50 spyware behaviors (vectors), collected by the long term monitor, using a tree classifier, an heuristic has been extracted. This heuristic shows that, in several cases, a spyware attack is identified by detecting a large number of `open` system calls, not related to a comparable amount of `read` and `write`. In the experiments on the Genome dataset, 8 spyware families, out of 13, have been successfully detected at run-time, whilst all of them (13 families on 13) have been correctly pre-filtered by the app-evaluator. We point out that actions of some spyware are hard to detect at run-time, as once they re-

ceive the correct authorizations (permissions), they perform operations which are legal on a behavioral point of view. Moreover, explicit spyware applications (i.e., non trojanized) can be found also on marketplaces, distributed with the specific purpose of sending device information to an external server. To this end, the app-evaluator is able to discern between a trojanized app and a genuine one [55], to overcome this weakness of the behavior-based global monitor.

Contagio Mobile

Contagio Mobile is a website that collects malware for several mobile OSes. Malware samples are submitted by readers, together with links to articles describing the malware. Differently from Genome, which has several samples for different pieces of malware, Contagio only presents few samples (generally one) for each malware family. Contagio collects malware since 2012, including also malicious apps found on Google Play, which may affect any kind of Android device, regardless of the nationality. We have tested MADAM against 18 malware families from the Contagio database. Detection results are reported in Table 5.6. Currently SMS trojan account for more than 90% of the total Android malicious apps which can be found in the wild [90]. In fact, as shown in Table 5.6, new malware families falling in this category can be found every year. SMS Trojans, results extremely effective since it cause a direct monetary loss to the user and is difficult to detect and stop. MADAM is effective against SMS trojans, detecting and stopping them at all cases. The testbed also show a malware belonging to the general trojan category. This malware hides inside an app describing Iranian recipes and only asks for the permission to access the SD card mass storage. The app maliciously exploits this permission, starting to overwrite any picture in the SD card with another image, permanently damaging the original files and slowly filling the memory. MADAM detects this misbehavior by mean of the long term classifier, which detects the change in the amount of `write` and `open`. This example shows the advantage of using an anomaly-based IDS, which is able to detect misbehaviors even if a policy has not been specified for it.

5.5.4 Usability Analysis

In this section we analyze the impact of MADAM on user perception and performance of the device.

Table 5.6: Analysis on malware from Contagio dataset. (**P** = pre-filtering, **R** = Runtime detection).

Malware	**Year**	**Type**	**P**	**R**
Dogowar	2011	SMS Trojan	√	√
FakeMart	2013	SMS Trojan	√	√
FakeNotify.B	2013	SMS Trojan	√	√
FakeRegSMS.B	2013	SMS Trojan	√	√
Fireye	2014	SMS Trojan	√	√
FlashFakeInstaller	2013	SMS Trojan	√	√
Geinimi	2011	SMS Trojan	√	√
GoogleFakeInstaller	2013	SMS Trojan	√	√
Lotoor	2011	Rootkit + Installer	√	√
Moghava	2012	Trojan	√	√
Oldboot.b	2012	Rootkit + Installer	√	√
OpFake	2012	SMS Trojan	√	√
Samsapo	2013	SMS Trojan	√	√
Scavir	2012	SMS Trojan	√	√
Selfmite.B	2013	SMS Trojan	√	√
Stiniter	2012	Rootkit + Installer	√	√
XXShenqi	2014	SMS Trojan	√	√
YZHC	2011	Rootkit + Installer	√	√

False Positives

False positives do not have a direct effect on the security of the device. However, they are not desirable since they require interaction with the user, who has to choose whether to remove or not the app deemed as malicious. Keeping their number low is thus of capital importance to avoid that the user prefers to deactivate MADAM. As shown in Table 5.3 and 5.4, the two K-NN classifiers of MADAM (short term and long term) have a FPR of 1,47% and 1.9%. Considering that in its operative conditions MADAM is handling 3660 events per hour, such an FPR would not be acceptable since it would generate about 30 FP per hour. However, the training set used for Table 5.3 and 5.4 is designed with different behaviors aimed at representing various possible usage patterns. During the real usage, we have experimentally verified that the behavioral variance is limited, i.e. the same or very similar behaviors are repeated for a long time, till a new event (e.g. users starts to interact with the device) changes the usage pattern. For this reason we have measured the real amount of false positives generated with three different

patterns of real device usage. As detailed in the following the false positives with a normal device usage average to 1 per day.

Usability experiments have been conducted on three devices with three users and different configurations. In the first experiment, called (i) *Light Usage*, we tested a Samsung Galaxy S2 with Android Jelly Bean version 4.2. The device only installed the native applications except for the super-user manager and MADAM. The user has been instructed to keep the smartphone mainly in standby, except for performing/receiving phone calls and/or sending/receiving text messages from the default messaging app. In the second experiment, called *Medium Usage*, we used a Samsung Galaxy Nexus with Android Jelly Bean version 4.3. The device installed 54 legitimate apps, including the native ones, MADAM and the super-user manager. The user has been instructed to use the device normally. The user on a daily basis accessed the Internet, three instant messaging applications and two social networks. Also he played daily with a 2D graphic videogame (`Angry Birds Space`) and a 3D ones (`Temple Run 2`) and took pictures with the smartphone camera. No new applications have been installed by this user on the device. In the thirs experiments, called *Heavy Usage*, we tested a LG Nexus 4 equipped with Android Kit-Kat version 4.4. At the beginning of the experiment the device equipped 52 apps including MADAM and super-user manager. The user has been instructed to keep the device always active (screen always on), interacting with it as much as possible. The user installed during the experiment 91 new legitimate applications during the tests and heavily used gaming apps, camera to take pictures and record video, in addition to instant messaging, text messages and phone calls.

The experiments lasted for one week, every day from 10:00 to 21:00, to avoid the reduction of activity normally caused by the night. Results are reported in Table 5.7. The table reports the total number of false positives issued during the three experiments by the two K-NN classifiers, and the average number per day. The FPR computed on the amount of monitored events by the two K-NN classifiers ($60 \times 60 \times 11 \times 7$ for the short term and $60 \times 11 \times 7$ for the long term one).

Table 5.7: False alarms experimental results.

Test	**FPs**	**FPR**	**FPs/day**
Light	3	$1 \cdot 10^{-5}$	0.5
Medium	8	$2.8 \cdot 10^{-5}$	1.1
Heavy	75	$2.6 \cdot 10^{-4}$	10.7

We have seen that if the device usage follows an average usage profile, the amount of false positives is at the threshold of 1 FP per day. It is worth noting that the number of FP per day noticeably raises with an heavy usage. In particular, a large number of false positives have been issued contemporary to the installation of new apps (the user installed 91 legitimate apps during the week). The event of installation of new apps is, in fact, computationally heavy and causes a sharp increase in the amount of issued system calls. However, the installation of a new app is handled by the package installer, which is a component of the operative system, thus MADAM will only notify an anomaly, without deeming any app as responsible for the misbehavior. Moreover, MADAM offers the possibility to handle false positives, allowing the users to re-train the classifiers, adding the false positives to the training set, as genuine eventsThanks to this functionality, the user can teach to MADAM its own behavior, reducing the amount of issued false positives. As a further experiments, retraining the classifiers adding 5 FPs from the third experiment to the training set and then using the device against in the same conditions of the same experiment, the FPs average per day reduced to 3. To further reduce the amount of issued FPs, it is also possible to put MADAM in "Training Mode". In this mode, the user will not be notified of any alarm, and the behaviors deemed as malicious are immediately added to the classifier training set. Note that for the k-NN classifier, this operation of adding a new element to the training set can be done simply adding the new element to its knowledge base, without training it again from scratches. This feature further motivate our decision in using the K-NN. However, "Training Mode" should be used carefully, i.e. the user should be sure that her device is not infected when activating this mode, to avoid the risk of training the classifier with malicious behavior as genuine. Moreover, over-training the classifier may cause over-fitting with a consequent detection performance degradation.

Performance Overhead

The performance overhead of MADAM has been measured through the `Quadrant Standard Edition` tool distributed as a free Android application through Google Play [5]. Performance tests have been performed on the same device used for malware detection experiments: Samsung Galaxy Nexus, CPU dual-core 1.2 GHz Cortex-A9, RAM 1GB, GPU PowerVR SGX540. The device runs Android 4.3 Jelly Bean, stock version. Apart from the native apps, the only app installed were the super user manager and the MADAM application.

[5] http://goo.gl/5ds6E7

Table 5.8 reports the benchmark for the system when MADAM was running (third column from left, “Vanilla”) and when it was not (second column left, “MADAM”). The last column reports the overhead computed as a percentage difference between the two performances. Benchmarks are provided as indexes, where a highest value means a better performance[6]. Benchmarks reported have been computed as the average of five experiments, both in “Vanilla” and “MADAM” configuration. The overhead of MADAM is caused by both the kernel module which hijacks system calls and a service which runs in background when the system is active. This service is responsible to handle the communication between the kernel level and the application level of MADAM. It also intercepts the events related to SMS and user activity, then classifies each monitored behavior. As shown, the performance impact of MADAM is acceptable; in fact, the overall performance impact (Total) is of 1.4%. It is worth noting that the stronger impact is on memory (9.4%). This is mainly due to the chosen classifier. In fact, the K-NN classifier, does not cause heavy load on the CPU. However, the K-NN requires to continuously keep in memory the whole training set, which may require a noticeable amount of space [86]. On the other hand, we note that MADAM has no impact on 2D/3D performances. This was expected, since MADAM functionalities does not influence the GPU. Furthermore, the 4% performance degradation on I/O is not perceived by user, whose experience is not altered [49].

Table 5.8: Benchmark Tests

Test	**Vanilla**	**MADAM**	**Overhead**
Total	2911	2868	1,4%
CPU	5509	5459	0,9%
Memory	2660	2409	9,4%
I/O	3860	3705	4%
2D	327	327	0%
3D	2250	2250	0 %

Energy Consumption

To measure the energy consumption of MADAM we have measured the difference in battery consumption over two periods of 24 hours, with and without MADAM. To apply the measurement, we used the Battery Monitor app [7]. Experiments have

[6] Units are not specified for each index.

[7] http://goo.gl/gg8hzH

been run on the same Samsung Galaxy Nexus used for the other experiments. The smartphone equips a 1750 mAh battery whose status reported at the beginning of the experiment by Battery Monitor is “good”. The smartphone has been kept in stand-by for the whole experiment, with no apps running, except for MADAM. The screen has been kept off for the whole experiment, network and localization interfaces were not active except for the 3G data connection. The results are reported in Figure 5.5. The graph of Figure 5.5 reports the two different discharges sampled

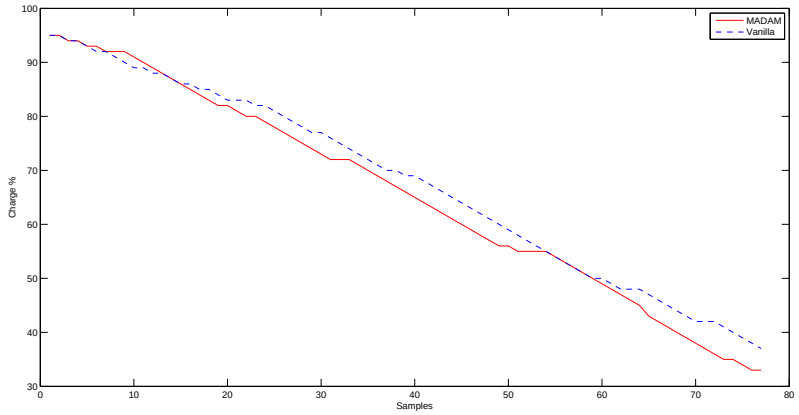


Figure 5.5: Energy impact evaluation of MADAM.

in a period of 24 hours with 77 sampling intervals (x-axis). The blue-dashed curve (Vanilla) representing the discharge without MADAM and the red-continuous one when MADAM is active. The monitoring started for both periods when the reported charge was at 95%. This is to avoid the recorded energy fluctuations happening immediately after the disconnection from the power supply. The distance between the two discharging curves is always lesser than 4%, which is the maximum value, recorded at the end of the monitoring period. The average consumption of the MADAM application reported by Battery Monitor is of 82 mAh, accounting to 4.6% of the total battery capacity. Thus, on a period of 24 hours, the MADAM user loses approximatively one hour of battery time, which is in line with current antivirus [2].

5.6 Conclusion

Starting from the end of 2011, attackers have directed their attention toward smartphones and tablets, producing and distributing hundreds of thousand of malicious

apps. These apps threaten the user data privacy, money and device integrity, being difficult to detect due to the low profile that they keep. This paper proposes MADAM, a multi-level host-based intrusion detection system for Android devices. MADAM enforces security, effectively protecting the device from malicious apps, detecting them at deploy-time and/or at runtime. MADAM has been tested against a testbed of 1300 malicious applications belonging to 40 malware families, showing an accuracy of 95% for runtime detection. The usage of MADAM does not modify the user experience, due to the low performance (1.4%) and energy overhead (4%) and to the very limited required user interaction.

MADAM has been designed to be totally extensible. In fact, MADAM's classifiers can learn new behaviors to adapt to different users. Also new heuristics can be added, to tackle specific misbehaviors. Given the fast malware evolution, this feature is of fundamental importance. For this reason, we plan as a future work the development of an interface to define new heuristics and security policies, which could be used also by the average user.

6

Conclusions

Security is of paramount importance on smartphones and tablets. In the last years, these new generation mobile devices have drawn the attention of attackers aiming at the user money, private information stored on the device, or at damaging the device itself. The main attack vector are trojanized apps, which are (i) easily distributed, through official or unofficial marketplaces, (ii) stealthy, since they offer the functionalities of real apps and perform the misbehavior in background, (iii) effective, since their effect may seriously compromise the user privacy, her money or the device.

Given the importance of protecting the user from threats brought by malicious apps, in this thesis we have presented a multi-component security framework for Android device protection. This framework combines the actions of four subcomponents which have been object of separate studies. Each subcomponent address a security issue or enforces a specific security abstraction. As discussed, the synergy of the four subcomponents successfully protects the device against malicious apps. The static analysis module MAETROID successfully detects all the analyzed malware, showing a FPR of 20% on a dataset of 12000 applications. On the other hand, the runtime monitor MADAM successfully detects 95% of the malicious applications coming from 40 malware families, with a false alarm rate of one FP per day. Joining the action of these two components, as proposed in Chapter 5, it is possible to block malicious apps at two different steps, i.e. deploy-time and runtime, with a low FPR and a very limited overhead on performances (1.4%). User acceptance has been measured with a survey distributed to 200 subjects.

The contract generator PICARD and the Probabilistic Security by Contract (SxCxP) give to the user the possibility to define probabilistic security policies. The SxCxP verifies at deploy time if the behavior of a new app will match the secu-

rity policies. If the app does not match the policy, the global monitor will avoid at runtime that the app will perform misbehavior in violation of the policy(es).

Thus, the proposed security framework is effective in protecting Android devices from malicious applications and at the same time allows to define custom security policies. The security policies may concern company environment, or may be related to user safety and protection(parental control, personal protection). The research of new policies is a direction for future works which may stem from this thesis. Android is being installed not only on mobile devices, but also as the operative system for Smart-Houses. House appliances, televisions, video-surveillance cameras and other non-mobile devices already install the Android OS, being able to exchange information, used for policies generally aimed at optimizing the energy consumption. If these systems become pervasive, it is likely that they will also drive the attention of attackers, as already happened for mobile devices. Extending the framework proposed in this thesis to address security in smart-house environments and, more generally, in Internet of Things architectures is a possible though challenging task. In fact, new security policies of the SxCxP module and MADAM heuristics should be engineered to consider not only the device (host-based), but a network of device and their interactions.

References

1. Contagio mobile, mobile malware mini dump.
2. How antivirus affect battery life. Last accessed on 23/02/2015.
3. Opengl es on android. http://developer.android.com/guide/topics/graphics/opengl.html.
4. 900 pay-per-call and other information services. `https://www.fcc.gov/guides/900-pay-call-and-other-information-services`, October 2014.
5. Fsecure mobile threat report q1 2014, 2014.
6. Global mobile statistics 2014 part a: Mobile subscribers; handset market share; mobile operators, 2014.
7. Global mobile statistics 2014 Part B: Mobile Web mobile broadband penetration; 3G/4G subscribers and networks. `http://goo.gl/REMiiQ`, 2014. Last access, Dec, 2014.
8. Mobile technology fact sheet, 2014.
9. Pandalabs quarterly report january-march 2014, 2014.
10. Sophos mobile security threat reports, 2014. Last Accessed: 20 November 2014.
11. Trojans are flipping over flappy bird, 2014.
12. A. Bose, K.G. Shin. Proactive Security For Mobile Messaging Networks. In *WiSe '06*, September 2006.
13. A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In ACM, editor, *1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM11)*, pages 3–14, 2011.
14. A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. Weiss. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, January 2011.
15. Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and Morley Mao, editors, *Security and Privacy in Communication Networks*, volume 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 86–103. Springer International Publishing, 2013.
16. SAP AG. Securing mobile apps in a byod world. `http://www.sap.com/bin/sapcom/en_us/downloadasset.2013-09-sep-06-08.Securing%20Mobile%20Apps%20in%20a%20BYOD%20World-pdf.html`, 2013.

17. A. Aldini, F. Martinelli, A. Saracino, and D. Sgandurra. A collaborative framework for generating probabilistic contracts. In *International Conference on Collaboration Technologies and Systems (CTS 2013)*, pages 139–142. IEEE, 2013.
18. Alessandro Aldini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. A collaborative framework for generating probabilistic contracts. In Waleed W. Smari and Geoffrey C. Fox, editors, *Proceedings of the 2013 IEEE International Conference on Collaboration Technologies and Systems*, SECOTS 2013, pages 139–143. IEEE Computer Society, 2013.
19. Alessandro Aldini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Detection of repackaged mobile applications through a collaborative approach. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2014.
20. Aniketos project. Available via `http://www.aniketos.eu/` on 19/09/2012, 2012.
21. A.P. Felt, E. Chin, S. Hanna, D. Song, D. Wagner. Android Permissions Demystified. In ACM, editor, *8th ACM conference on Computer and Communications Security (CCS'11)*, pages 627–638, 2011.
22. Appthority. Appthority trust score. `https://www.appthority.com/services/app-reputation-app-trust-score`, 2014.
23. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.
24. Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp Styp-Rekowsky. AppGuard – Enforcing User Requirements on Android Apps. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, pages 543–548. Springer Berlin Heidelberg, 2013.
25. Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard – fine-grained policy enforcement for untrusted android applications. In Joaquin Garcia-Alfaro, Georgios Lioudakis, Nora Cuppens-Boulahia, Simon Foley, and William M. Fitzgerald, editors, *Data Privacy Management and Autonomous Spontaneous Security*, Lecture Notes in Computer Science, pages 213–231. Springer Berlin Heidelberg, 2014.
26. Christel Baier, Bettina Engelen, and Mila Majster-Cederbaum. Deciding bisimilarity and similarity for probabilistic processes. *Journal of Computer and System Sciences*, 60(1):187–231, 2000.
27. Rajeev Bhattacharya, Timothy M Devinney, and Madan M Pillutla. A formal model of trust based on outcomes. *Academy of management review*, 23(3):459–472, 1998.
28. N. Bielova and F. Massacci. Predictability of enforcement. In *Proceedings of the International Symposium on Engineering Secure Software and Systems 2011*, volume 6542, pages 73–86. Springer, 2011.
29. A. Bogliolo, P. Polidori, A. Aldini, W. Moreira, P. Mendes, M. Yildiz, C. Ballester, and J.-M. Seigneur. Virtual currency and reputation-based cooperation incentives in user-centric networks. In *Proc. of Wireless Communications and Mobile Computing Conference (IWCMC-2012)*, pages 895–900. IEEE, 2012.
30. Abhijit Bose and Kang G. Shin. Proactive security for mobile messaging networks. In *WiSe '06: Proceedings of the 5th ACM workshop on Wireless security*, pages 95–104, New York, NY, USA, 2006. ACM.

31. Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.
32. Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 51–62, New York, NY, USA, 2011. ACM.
33. Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 131–146, Berkeley, CA, USA, 2013. USENIX Association.
34. I. Burguera, U. Zurutuza, and S. Nadijm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *SPSM'11*. ACM, October 2011.
35. Troutman C. Say goodbye to custom "stock" roms and hello to xposed framework, May 2013.
36. Sung-Hyuk Cha. Comprehensive survey on distance/similarity measures between probability density functions. *International Journal of Mathematical Models and Methods in Applied Sciences*, 1(4):300–307, 2007.
37. Maria Garnaeva Christian Funk. Kaspersky security bullettin 2013, December 2013.
38. Filip Chytry. How are you doing mr. android?, January 2014.
39. Samsung Electronics Co. White paper: Samsung knox premium. `https://www.samsungknox.com/en/system/files/whitepaper/files/White_Paper_Samsung_KNOX_Premium%20_V1_20140919_0.pdf`, September 2014.
40. Alessandro Colantonio. Prioritizing Role Engineering Objectives Using the Analytic Hierarchy Process. In *Information Systems: Crossroads for Organization, Management, Accounting and Engineering*, pages 419–427. Physica-Verlag HD, 2012.
41. M. Conti, V.T.N. Nguyen, and B. Crispo. CRePE: context-related policy enforcement for android. In *ISC'10 Proceedings of the 13th international conference on Information security* , pages 331–345. Springer-Verlag, 2010.
42. G. Costa, N. Dragoni, V. Issarny, A. Lazouski, F. Martinelli, F. Massacci, I. Matteucci, and R. Saadi. Security-by-Contract-with-Trust for mobile devices. *JOWUA*, 1(4):75–91, Dec. 2010.
43. G. Costa, N. Dragoni, A. Lazouski, F. Martinelli, F. Massacci, and I. Matteucci. Extending Security-by-Contract with quantitative trust on mobile devices. In *Proceeding of the Fourth International Conference on Complex, Intelligent and Software Intensive Systems*, pages 872–877. IEEE Computer Society, 2010.
44. Costantino, G. and Martinelli, F. and Petrocchi, M. Priorities-based review computation. In *AAAI Spring Symposium, 2012 1st Workshop on Intelligent Web Services meet Social Computing*, volume SS-12-04, 2012.
45. D. Barrera, H.G. Kayacik, P.C. van Oorschot, A. Somayaji. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, October 2010.
46. D. Damopoulos, S.A. Menesidou, G. Kambourakis, M. Papadaki, N. Clarke, S. Gritzalis. Evaluation of Anomaly-Based IDS for Mobile Devices Using Machine Learning Classifiers. *Security and Communications Networks*, 5(00):1–9, 2011.

47. D. Mutz, F. Valeur, G. Vigna. Anomalous System Call Detection. *ACM Transactions on Information and System Security*, 9(1):61–93, February 2006.
48. DAI-Labor. Androlyzer. `https://www.androlyzer.com/`, 2014.
49. Michael J. Darnell. Acceptable system response times for tv and dvr. In *Proceedings of the 5th European Conference on Interactive TV: A Shared Experience*, EuroITV'07, pages 47–56, Berlin, Heidelberg, 2007. Springer-Verlag.
50. B. Delahaye, B. Caillaud, and A. Legay. Probabilistic contracts: A compositional reasoning methodology for the design of stochastic systems. In *10th International Conference on Application of Concurrency to System Design (ACSD), 2010*. IEEE, 2010.
51. Benoît Delahaye and Benoît Caillaud. A model for probabilistic reasoning on assume/guarantee contracts. arXiv preprint arXiv:0811.1151, 2008.
52. Benoît Delahaye, Benoît Caillaud, and Axel Legay. Probabilistic contracts: a compositional reasoning methodology for the design of systems with stochastic and/or non-deterministic aspects. *Formal Methods in System Design*, 38(1):1–32, 2011.
53. Josée Desharnais, François Laviolette, and Mathieu Tracol. Approximate analysis of probabilistic processes: Logic, simulation and games. In *Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, QEST '08, pages 264–273, Washington, DC, USA, 2008. IEEE Computer Society.
54. G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. MADAM: A Mult-Level Anomaly Detector for Android Malware. In *6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security, MMM-ACNS 2012, St. Petersburg, Russia*, volume 7531 - 2012. Springer Verlag, October 2012.
55. Gianluca Dini, Fabio Martinelli, Ilaria Matteucci, Marinella Petrocchi, Andrea Saracino, and Daniele Sgandurra. A Multi-Criteria-Based Evaluation of Android Applications. Technical report, Istituto di Informatica e Telematica, CNR, Pisa, 2012. http://www.iit.cnr.it/node/17019,Last access, 27th of Aug, 2014.
56. Gianluca Dini, Fabio Martinelli, Ilaria Matteucci, Marinella Petrocchi, Andrea Saracino, and Daniele Sgandurra. A multi-criteria-based evaluation of android applications. In ChrisJ. Mitchell and Allan Tomlinson, editors, *Trusted Systems*, volume 7711 of *Lecture Notes in Computer Science*, pages 67–82. Springer Berlin Heidelberg, 2012.
57. Gianluca Dini, Fabio Martinelli, Ilaria Matteucci, Marinella Petrocchi, Andrea Saracino, and Daniele Sgandurra. A multi-criteria-based evaluation of android applications. In *Trusted Systems, 4th International Conference, INTRUST 2012, London, UK, December 17-18, 2012. Proceedings*, pages 67–82, 2012.
58. Gianluca Dini, Fabio Martinelli, Ilaria Matteucci, Andrea Saracino, and Daniele Sgandurra. Evaluating the Trust of Android Applications through an Adaptive and Distributed Multi-criteria Approach. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2013.*, pages 1541–1546, 2013.
59. Gianluca Dini, Fabio Martinelli, Ilaria Matteucci, Andrea Saracino, and Daniele Sgandurra. Introducing Probabilities in Contract-Based Approaches for Mobile Application Security. In *Data Privacy Management and Autonomous Spontaneous Security*, LNCS, pages 284–299. Springer Berlin Heidelberg, 2014.
60. Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Madam: A multi-level anomaly detector for android malware. In Igor Kotenko and Victor Skormin, editors, *Computer Network Security*, volume 7531 of *Lecture Notes in Computer Science*, pages 240–253. Springer Berlin Heidelberg, 2012.

61. Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. MADAM: A Multi-level Anomaly Detector for Android Malware. In *Proceedings of the 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security: Computer Network Security*, MMM-ACNS'12, pages 240–253. Springer-Verlag, 2012.
62. Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Madam: A multi-level anomaly detector for android malware. In Igor Kotenko and Victor Skormin, editors, *Computer Network Security*, volume 7531 of *Lecture Notes in Computer Science*, pages 240–253. Springer Berlin / Heidelberg, 2012.
63. Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Probabilistic contract compliance for mobile applications. In *ARES*, pages 599–606. IEEE Computer Society, 2013.
64. N. Dragoni, F. Martinelli, F. Massacci, P. Mori, C. Schaefer, T. Walter, and E. Vetillard. Security-by-contract (SxC) for software and services of mobile systems. In *At your service - Service-Oriented Computing from an EU Perspective.* MIT Press, 2008.
65. N. Dragoni and F. Massacci. Security-by-contract for web services. In *SWS*, pages 90–98, 2007.
66. Arvind Easwaran, Sampath Kannan, and Insup Lee. Optimal control of software ensuring safety and functionality. Technical Report MS-CIS-05-20, University of Pennsylvania, 2005.
67. William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.
68. William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 57(3):99–106, March 2014.
69. William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245, New York, NY, USA, 2009. ACM.
70. Adrienne Porter Felt, Serge Egelman, and David Wagner. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 33–44, New York, NY, USA, 2012. ACM.
71. Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: user attention, comprehension, and behavior. In *Symposium On Usable Privacy and Security, SOUPS '12, Washington, DC, USA - July 11 - 13, 2012*, page 3, 2012.
72. A. Fernandes, E. Kotsovinos, S. Ostring, and B. Dragovic. Pinocchio: Incentives for honest participation in distributed trust management. In *Proc. of iTrust'04*, volume 2995 of *LNCS*, pages 63–77. Springer, 2004.
73. V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.

74. J. Forristal. One root to own them all. https://media.blackhat.com/us-13/US-13-Forristal-Android-One-Root-to-Own-Them-All-Slides.pdf,{L}astaccess,27thofAug,2014, 2013.
75. G.A. Jacoby, R. Marchany, N.J.Davis, IV. How Mobile Host Batteries Can Improve Network Security. *IEEE Security and Privacy*, 4:40–49, 2006.
76. O. Gadyatskaya, F. Massacci, and A. Philippov. Security-by-Contract for the OSGi Platform. In *IFIP TC 11 Information Security and Privacy Conference*, pages 364–375, 2012.
77. P. Greci, F. Martinelli, and I. Matteucci. A framework for contract-policy matching based on symbolic simulations for securing mobile device application. In *ISoLA*, pages 221–236, 2008.
78. Holger Hermanns, Augusto Parma, Roberto Segala, Björn Wachter, and Lijun Zhang. Probabilistic logical characterization. *Inf. Comput.*, 209(2):154–172, February 2011.
79. X.A. Hoang and J. Hu. An efficient hidden Markov model training scheme for anomaly intrusion detection of server applications based on system calls . In *12th IEEE International Conferecence On Networks, ICON 2004*, volume 2, pages 470–474. IEEE, November 2004.
80. Google Inc. Google verify app. https://support.google.com/accounts/answer/2812853?hl=en. Last access, Dec, 2014.
81. Ryan Johnson and Angelos Stavrou. Forced-path execution for android applications on x86 platforms. In *Proceedings of the 2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, SERE-C '13, pages 188–197, Washington, DC, USA, 2013. IEEE Computer Society.
82. Audun Josang. Trust-based decision making for electronic transactions. In *Proceedings of the Fourth Nordic Workshop on Secure Computer Systems (NORDSEC'99)*, pages 496–502, 1999.
83. Juniper Networks Global Threat Center. Malicious Mobile Threats Report 2010/2011, 2011.
84. Radu Jurca, Florent Garcin, Arjun Talwar, and Boi Faltings. Reporting incentives and biases in online review forums. *ACM Trans. Web*, 4(2):5:1–5:27, April 2010.
85. A.P. Koresow. Intrusion detection via system call traces. *Software*, 14(5), 1997.
86. O. Kramer. Dimensionality reduction by unsupervised k-nearest neighbor regression. In *Machine Learning and Applications and Workshops (ICMLA), 2011 10th International Conference on*, volume 1, pages 275–278, Dec 2011.
87. Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *1st IEEE International Conference on Data Mining*, page 313. IEEE, 2001.
88. M. La Polla, F. Martinelli, and D. Sgandurra. A survey on security for mobile devices. *Communications Surveys Tutorials, IEEE*, PP(99):1 –26, 2012.
89. Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. A survey on security for mobile devices. *Communications Surveys Tutorials, IEEE*, 15(1):446 –471, quarter 2013.
90. Kindsight Security Labs. Kindsight security labs malware report – h1 2014, 2014.
91. Yepang Liu and Chang Xu. Veridroid: Automating android application verification. In *Proceedings of the 2013 Middleware Doctoral Symposium*, MDS '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.
92. M. Nauman, S. Khan, X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In ACM, editor, *5th ACM Sym-*

posium on Information Computer and Communication Security (ASIACCS'10), April 2010.
93. F. Maggi, M. Matteucci, and S. Zanero. Detecting Intrusions through System Call Sequence and Argument Analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4), October-December 2010.
94. Fabio Martinelli and Charles Morisset. Quantitative access control with partially-observable markov decision processes. In *Proceedings of CODASPY '12*, pages 169–180. ACM, 2012.
95. Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Classifying android malware through subgraph mining. In *DPM/SETOP*, pages 268–283, 2013.
96. Ilaria Matteucci, Paolo Mori, and Marinella Petrocchi. Prioritized execution of privacy policies. In *Data Privacy Management (DPM), 2012 7th Intl. Workshop on*, pages 133–145, 2012.
97. Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, November 2012.
98. V. Misra. What are the exact mechanisms/flaws exploited by the "rage against the cage" and "z4root" android exploits?
99. George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, January 1997.
100. M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 340–349, dec. 2009.
101. Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 241–252, New York, NY, USA, 2012. ACM.
102. R. L. Plackett. Karl Pearson and the Chi-Squared Test. *International Statistical Review / Revue Internationale de Statistique*, 51(1):59–72, 1983.
103. R. Cannings. An update on Android Market security, 2011. http://googlemobile.blogspot.com/2011/03/update-on-android-market-security.html.
104. L. Rajbhandari and E.A. Snekkenes. An approach to measure effectiveness of control for risk analysis with game theory. In *Socio-Technical Aspects in Security and Trust (STAST), 2011 1st Workshop on*, pages 24 –29, 2011.
105. Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on System Security (EUROSEC)*, Prague, Czech Republic, April 2013.
106. Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.
107. S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.R. Sadeghi, B. Shastry. Practical and Lightweight Domain Isolation on Android. In ACM, editor, *1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM11)*, pages 51–61, 2011.

108. T. L. Saaty. A scaling method for priorities in hierarchical structures. *Journal of Mathematical Psychology*, 15(3):234–281, 1977.
109. T. L. Saaty. How to make a decision: The analytic hierarchy process. *European Journal of Operational Research*, 48(1):9–26, 1990.
110. T. L. Saaty. Decision-making with the AHP: Why is the principal eigenvector necessary. *European Journal of Operational Research*, 145(1):85–91, 2003.
111. T. L. Saaty. Decision making with the analytic hierarchy process. *International Journal of Services Sciences*, 1:83–98, 2008.
112. Aubrey-Derrick Schmidt, Rainer Bye, Hans-Gunther Schmidt, Jan Hendrik Clausen, Osman Kiraz, Kamer Ali Yüksel, Seyit Ahmet Çamtepe, and Sahin Albayrak. Static Analysis of Executables for Collaborative Malware Detection on Android. In *Proceedings of IEEE International Conference on Communications, ICC 2009, Dresden, Germany, 14-18 June 2009*, pages 1–5. IEEE, 2009.
113. Aubrey-Derrick Schmidt, Frank Peters, Florian Lamour, Christian Scheel, Seyit Ahmet Çamtepe, and Sahin Albayrak. Monitoring smartphones for anomaly detection. *Mob. Netw. Appl.*, 14(1):92–106, 2009.
114. R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 15–28, 2003.
115. Michael Ian Sharkey. *Probabilistic Proof-carrying Code*. PhD thesis, Carleton University, 2012.
116. Michael Spreitzenbarth, Thomas Schreck, Florian Echtler, Daniel Arp, and Johannes Hoffmann. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, pages 1–13, 2014.
117. M. Srivatsa, L. Xiong, and L. Liu. Trustguard: countering vulnerabilities in reputation management for decentralized overlay networks. In *Proc. of WWW'05*, pages 422–431. ACM Press, 2005.
118. T. M. Cover, P.E. Hart. Nearest Neighbor Pattern Classification. *IEEE Transactions on Information Theory*, IT-13(1):21–27, January 1967.
119. The Organisation for Economic Co-operation and Development (OECD). The OECD Privacy Framework, 2013. http://goo.gl/cQLB4X. Last access: 27th of July, 2014.
120. TrustGo. Trustgo website. `http://www.trustgo.com/en/`. Last access, 27th of Aug, 2014.
121. Yasuyuki Tsukada. Interactive and probabilistic proof of mobile code safety. *Automated Software Engineering*, 12(2):237–257, 2005.
122. IDC Corporate USA. Worldwide quarterly mobile phone tracker. `http://goo.gl/NxMZw.{L}astaccess,27thofAug,2014`, 2013.
123. Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. *SIGMETRICS Perform. Eval. Rev.*, 42(1):221–233, June 2014.
124. Giovanni Vigna and Richard A. Kemmerer. Netstat: A network-based intrusion detection system. *J. Comput. Secur.*, 7(1):37–71, January 1999.
125. Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 21–30, New York, NY, USA, 2004. ACM.

126. W. Enck, M. Ongtang, P. McDaniel. On Lightweight Mobile Phone Application Certification. In ACM, editor, *16th ACM conference on Computer and Communications Security (CCS'09)*, pages 235–254, November 2009.
127. Webroot. Brightcloud mobile app reputation service. `http://www.brightcloud.com/services/mobile-app-reputation.php`, 2014.
128. Liang Xie, Xinwen Zhang, Jean-Pierre Seifert, and Sencun Zhu. pBMDS: a behavior-based malware detection system for cellphone devices. In *Proceedings of the Third ACM Conference on Wireless Network Security, WISEC 2010, Hoboken, New Jersey, USA, March 22-24, 2010*, pages 37–48. ACM, 2010.
129. Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.
130. Xuxian Jiang. Multiple Security Alerts: New Android Malware Found in Official and Alternative Android Markets, 2011. http://www.csc.ncsu.edu/faculty/jiang/pubs/index.html.
131. Y. Zhou, X. Zhang, X. Jiang, V. W. Freeh. Taming information-stealing smartphone applications (on android). In *4th International Conference on Trust and Trustworthy Computing (TRUST 2011)*, June 2011.
132. Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.
133. M. Yang, Q. Feng, Y. Dai, and Z. Zhang. A multi-dimensional reputation system combined with trust and incentive mechanisms in p2p file sharing systems. In *Proc. of IEEE Distributed Computing Systems Workshops*, 2007.
134. Nong Ye and Qiang Chen. An anomaly detection technique based on a chi-square statistic for detecting intrusions into information systems. *Quality and Reliability Engineering International*, 17(2):105–112, 2001.
135. Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.
136. Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.
137. Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST'11, pages 93–107, Berlin, Heidelberg, 2011. Springer-Verlag.

Acknowledgments

I would like to thank **Dr. Fabio Martinelli** and **Prof. Gianluca Dini**, which have been more than simply my co-advisors in this 3 years experience. In particular I'd like to express my gratitude to Dr. Fabio Martinelli for supporting me and teaching me from scratch how is research and how to defend the soundness of my ideas also against the hardest reviewer. Thanks to Prof. Gianluca Dini, who advised me also for my Master thesis and that has always been my model, inspiring me in taking an academic career and passing me the passion for teaching.

Thanks to **Dr. Sajal K. Das**, for hosting me at Missouri University of Science and Technology for my visiting period and for all the precious advises he gave me on how keeping to the highest standard my research work.

My greatest gratitude goes to **Daniele Sgandurra**[1], who taught me how to write scientific papers, being extremely patient in correcting me all my mistakes. Thanks for your guidance, starting from the first days of my Ph.D. and continuing also now that you are in London.

Thanks to **Ilaria Matteucci**, for introducing to this engineer, the basis of logic and theoretical computer science. Also thanks to bear my stubbornness and for continuously pushing me to hurry up with this thesis.

Thanks to **Fabio Del Bene**, for always listening at my ideas and for his irreplaceable help in implementing them.

A special thank to **Domenico De Guglielmo** for sharing and discussing with me so many ideas in these years.

[1] No means to disrispect, but to make it more personal, I'm cutting the Dr. and Prof. stuff from now on.

Thanks (and tanks :P) to **Francesco Restuccia**, thatm, no matter what, he will always be my friend and great colleague.

Many thanks to **Artsiom Yautsiukhin, Sasha Lazouski** and **Leanid Krautsevich**, my colleagues from Belarus who shared with me the B64 room (listening at my continuous ravings) and/or very good drinks in Munich, Bertinoro and wherever we have been together.

Thanks to **Gianpiero Costantino**, **Davide D'Arenzo** and **Gaetano Mancini**, for making my permanence at CNR happier and funnier.

Thanks to all my colleagues from the Dipartimento di Ingegneria dell'Informazione for advises and/or nice time spent together: **Roberta Daidone, Marco Tiloca, Francesco Giurlanda, Pericle Perazzo, Davide Gazzé, Giovanni Virdis**.

Thanks to my friend and advisor **Sara Lioba Volpi**. I'd never make it till here if had not met you five years ago. Thanks for all.

My thanks also go to my **family**. My parents have always pushed me to reach the highest standard, even if at the beginning was hard for them to understand why I was doing a Ph.D..

A special thanks to **PHD Comics** and **Zerocalcare**, for the daily support that their comics gave to my Ph.D. and also for all the pictures I have borrowed (always credited) for my lessons and presentations.

A lot of thanks to my friends from the archery club **Frecce Pisane**, who so many times accepted "I'm writing my thesis" as a reason for "I cannot come shooting today". In particular I'd like to thank **Paolo Petri** and **Luca Caprili** which are the real core of my archery club and **Corrado Chiaramida** and **Virginia Muzi** who draw me to my greatest passion (after research).

Thanks, thanks, really a lot of thanks to my greatest friends **Sara "Paio" Paioletti** and **Ilaria Fanizzi** for their support in the darkest period of my Ph.D. The restaurant club will never die.

Thanks (mamnoon) to **Mina Alishahi**, for believing in me more than myself and for loving me back to life.